KSConf Documentation

Release 0.13.1

Lowell Alleman

Oct 05, 2023

CONTENTS

Author
Lowell Alleman (Kintyre)

Version
0.13

CHAPTER

ONE

WELCOME TO KSCONF!

KSCONF is a modular command line tool for Splunk admins and app developers. It's quick and easy to get started with basic commands and grow into the more advanced commands as needed. Thank you for reviewing our expanding body of documentation to help smooth your transition to a more well-managed Splunk environment and explore ways to integrate Ksconf capabilities into your existing workflow.

We are glad you are here! No matter where you're starting from, Ksconf can help. Let us know if there is anything we can do to help along your journey.

– Kintyre, a CDI Company

CHAPTER

TWO

INSTALL

Ksconf can be directly installed as a Python (via `pip`) or as a Splunk app. The Splunk app option is often easier.

To install as a **python package**, run the following:

```
pip install ksconf
```

To install the **Splunk app**, download the latest KSCONF App for Splunk release. Note that a one-time registration command is needed to make `ksconf` executable:

```
splunk cmd python3 $SPLUNK_HOME/etc/apps/ksconf/bin/install.py
```

CHAPTER

THREE

USER GUIDE

3.1 Introduction

KSCONF (Ksconf Splunk Configuration tool) is a command-line tool that helps administrators and developers manage their Splunk environments by enhancing their ability to control configuration files. By design, the interface is modular so that each function (aka subcommand) can be learned quickly and used independently. Most Ksconf commands are simple enough for a quick one-off job, yet reliable enough to integrate into complex app build and deployment workflow.

Ksconf helps manage the nuances of storing Splunk apps in a version control system, such as git. It also supports pointing live Splunk apps to a working tree, merging changes from the live system's (local) folder to the version controlled folder (often 'default'), and in more complex cases, it deals with more than one *layer* of "default", which Splunk can't handle natively.

Note: What KSCONF is not

Ksconf does *not* replace your existing Splunk deployment mechanisms or version control tools. The goal is to complement and extend, not replace, the workflow that works for you.

3.1.1 Design principles

Ksconf is a toolbox.
: Each tool has a specific purpose and function that works independently. Borrowing from the Unix philosophy, each command should do one thing well and be easily combined to handle higher-order tasks.

When possible, be familiar.
: Various commands borrow from popular UNIX command line tools such as **`grep`** and **`diff`**. The modular nature of the command and other design features were borrowed from **`git`** and **`splunk`** as well.

Don't impose workflow.
: Ksconf works with or without version control and independently of your deployment mechanisms. If you are looking to implement these things, Ksconf is a great building block.

Embrace automated testing.
It's impractical to check every scenario between each release, but significant work has gone into unit testing the CLI to avoid breakage.

3.1.2 Common uses for Ksconf

- Build and package Splunk apps
- Promote changes from `local` to `default`
- Maintain multiple independent layers of configurations
- Reduce duplicate settings in a local file
- Upgrade apps stored in version control
- Merge or separate configuration files
- Git pre-commit hook for validation
- Git post-checkout hook for workflow automation
- Send *.conf* stanzas to a REST endpoint (Splunk Cloud or no file system access)

3.1.3 Getting started

You're in the right place. If you are a beginner, try checking these out first:

- *Cheat Sheet* - Like jumping in the deep end, or prefer examples of descriptions? Start here.
- *Concepts* - To get a more theoretical background on why these things matter.
- *Commands* - Start here if you'd like a more thorough introduction.

3.2 Concepts

3.2.1 Configuration layers

The idea of configuration layers is shared across multiple actions in Ksconf. Specifically, *combine* is used to merge multiple layers, and the *unarchive* command can be used to install or upgrade an app in a layer-aware way.

What's the problem?

In a typical enterprise deployment of Splunk, a single app can easily have multiple logical sources of configuration:

1. Upstream app releases, often from Splunkbase
2. Organization-specific customizations or fixes added by a local developer
3. Fixes to buggy upstream settings, like `indexes.conf`, requested by your Splunk admin

4. Custom knowledge objects created by subject matter experts

Ideally we would like to version control these, but doing so is complicated because normally you have to manage all four of these logical layers in one 'default' folder.

Note: Isn't that what the **local** folder is for?

Splunk requires that app settings be located either in `default` or `local`; and managing local files with version control leads to merge conflicts. So effectively, all version controlled settings need to be in `default`, or risk merge conflicts. However, making changes to the `default` folder causes issues when you attempt to upgrade an app upstream. See how this is a dilemma?

Let's suppose a new upstream version is released. If you aren't managing layers independently, then you have to manually upgrade the app, being careful to preserve all custom configurations. Compare this to the solution provided by the *combine* functionality. The layered approach provides an advantage because logical sources can be stored separately in their own directories, thus allowing them to be modified independently. Using this approach, changes in the "upstream" layer will only come from an official release, and the organizational layer will contain customizations made solely by your organization. Practically, this means it's no longer necessary to comb through commit logs identifying which custom changes need to be preserved and reapplied.

While this doesn't completely remove the need for a person to review app upgrades, it does lower the overhead enough that updates can be pulled in more frequently, thus minimizing divergence.

3.2.2 Minimizing files

A typical scenario:

To customize a Splunk app or add-on, many admins simply copy the conf file from default to local and then apply changes to the local copy. That's a common practice, but stopping there complicates future upgrades. The next step should be to clean up the local file, deleting all the unmodified entries that were copied from default.

Why does this matter?

If you've copied a default file into the local folder, this means that local file doesn't contain *only* your settings, it contains a copy of *all* of the default settings too. So in the future, fixes published by the app creator are likely to be masked by your local settings. A better approach is to reduce the local conf file leaving only the stanzas and settings that you intended to change. While this is a monotonous to do by hand, it is easily accomplished by *ksconf minimize*. This makes your conf files easier to read and simplifies upgrades.

What does Splunk have to say about this? (From the docs)

> "When you first create this new version of the file, **start with an empty file and add only the attributes that you need to change.** Do not start from a copy of the default directory. If you copy the entire default file to a location with higher precedence, any changes to the default values that occur through future Splunk Enterprise upgrades cannot take effect, because the values in the copied file will override the updated values in the default file." – [SPLKDOC1]

Tip:

It's a good practice to minimize your files right away. If you wait, it may not be obvious what specific version of default that local was copied from. In other words, if you run the **`minimize`** command *after* you've upgraded the default folder, you may need to do extra work to manually reconcile upgrade differences, because any changes made between the initial version of the default file and the most recent release of the conf file cannot, be automatically addressed in this fashion. If your files are all in git, and you know a REF of the previous version of your default file, you can use some commands like this:

```
# Review the output of the log, and find the revision of the last change
git log --oneline -- default/inputs.conf

# Assuming "e633e6" was identified as the desired baseline ref, based on␣
↪the 'log' output

# Compare what's changed in the 'inputs.conf' file between releases (FYI␣
↪only)
ksconf diff <(git show e633e6:./default/inputs.conf) default/inputs.conf

# Now apply the 'minimization' based on the original version of inputs.conf
ksconf minimize --target=local/inputs.conf <(git show e633e6:./default/
↪inputs.conf)
```

As always, be sure to double check the results.

3.3 Installation Guide

KSCONF can be installed either as a Splunk app or a Python package. Picking the option that's right for you is fairly easy.

Unless you have experience with Python packaging or are planning on customizing or extending Ksconf, then the *Splunk app* is likely the best place for you to start. The native Python package works well for many developer-centric scenarios, but installation ends up being complicated for the more typical admin-centric use-case. Therefore, most users will find it easier to start with the Splunk app.

Note: The introduction of a Splunk app is a fairly new occurrence (as of the 0.6.x release). Originally we resisted this idea, since `ksconf` was designed to manage other apps, not live within one. Ultimately however, the packaging decision was made to ensure users of all levels can utilize the program, as Python packaging is a mess and can be daunting for the uninitiated.

3.3.1 Overview

Install	Advantages	Potential pitfalls
Python package	• Most 'pure' and flexible install • One command install. (ideal) • Easy upgrades • More extendable (plugins) • *Install Python package*	• Lots of potential variations and pitfalls • Many Linux distros don't ship with `pip` • Must consider/coordinate installation user. • Often requires some admin access. • Too many install options (complexity)
Splunk app	• Quick installation (single download) • Requires one time bootstrap command • Self contained; no admin access require • Fast demo; fight with `pip` later • *Install Splunk App*	• Crippled Python install (no `pip`) • Can't add custom extensions or plugins • No CLI completion (yet) • *Grandfather Paradox*
Offline package	• Security: strict review and change control • *Advanced Installation Guide*.	• Requires many steps. • Inherits 'Python package' pitfalls.

3.3.2 Requirements

Python package install:

- Python Supports Python 3.7+
- PIP
- Tested on Mac, Linux, and Windows

Splunk app install:

- Splunk 8.0 or greater is installed

3.3.3 Install Splunk App

Download and install the KSCONF App for Splunk. Then open a shell, switch to the Splunk user account and run this one-time bootstrap command.

```
splunk cmd python3 $SPLUNK_HOME/etc/apps/ksconf/bin/install.py
```

On Windows, open a terminal as Administrator and type:

```
cd "C:\Program Files\Splunk"
bin\splunk.exe cmd python3 etc\apps\ksconf\bin\install.py
```

This will add `ksconf` to Splunk's `bin` folder, thus making it executable either as `ksconf` or, less optimally, `splunk cmd ksconf`. (If you can run `splunk` without giving it a path, then `ksconf` should work too.)

At some point we may add an option for you to do this setup step from the UI.

Note: Alternate download

You can also download the latest (and pre-release) SPL from the GitHub Releases page. Download the file named like `ksconf-app_for_splunk-ver.tgz`

3.3.4 Install Python package

Quick Install

Using pip:

```
pip install ksconf
```

System-level install: (For Mac/Linux)

```
curl https://bootstrap.pypa.io/get-pip.py | sudo python - ksconf
```

Enable Bash Completion

Context-aware autocomplete can be a great time saver. If you're on a Mac or Linux, and would like to enable bash completion, run these commands:

```
pip install argcomplete
echo 'eval "$(register-python-argcomplete ksconf)"' >> ~/.bashrc
```

(This option is not currently available for Splunk App installs due to a lack of documentation and testing available presently. It should be possible. Pull requests are welcome.)

Ran into issues?

If you encounter any issues, please refer to the *Advanced Installation Guide*. Substantial time and effort was placed into the assembly of the information based on various scenarios we encountered. A good place to begin would be in the *Troubleshooting* section.

3.3.5 Install from GIT

If you'd like to contribute to ksconf, or just build the latest and greatest, then installing from the git repository is a good choice. (Technically this is still installing with `pip`, so it's easy to switch between a PyPI install, and a local install.)

```
git clone https://github.com/Kintyre/ksconf.git
cd ksconf
pip install .
```

See *Developer setup* for additional details about contributing to ksconf.

3.3.6 Validate the install

No matter how you install `ksconf`, you can confirm that it's working with the following command:

```
ksconf --version
```

The output should look something like this:

```
                                #
                                ##
 ###  ##     #### ###### #######  ###  ##  #######
 ### ##     ###  ###           ## #### ##
 #####      ###  ###      ##   ## #######  #######
 ### ##     ###  ###      ##   ## ### ###  ##
 ###  ## #####    ######   #####  ###  ##  ##
                                     #

ksconf 0.7.3  (Build 376)
Python: 2.7.15  (/Applications/splunk/bin/python)
Git SHA1 dc94f811 committed on 2019-06-05
Installed at: /Applications/splunk/etc/apps/ksconf/bin/lib/ksconf
Written by Lowell Alleman <lowell@kintyre.co>.
Copyright (c) 2019 Kintyre Solutions, Inc, all rights reserved.
Licensed under Apache Public License v2

  kintyre_splunk_conf  (0.7.3)

    Commands:
```

(continues on next page)

(continued from previous page)

```
    check           (stable)    OK
    combine         (beta)      OK
    diff            (stable)    OK
    filter          (alpha)     OK
    merge           (stable)    OK
    minimize        (beta)      OK
    promote         (beta)      OK
    rest-export     (beta)      OK
    rest-publish    (alpha)     OK   (splunk-sdk 1.6.6)
    snapshot        (alpha)     OK
    sort            (stable)    OK
    unarchive       (beta)      OK
    xml-format      (alpha)     OK   (lxml 4.2.5)
```

Missing 3rd party libraries

Note: *Splunk app for KSCONF* users don't need to worry about this.

As of version 0.7.0, ksconf now includes commands that require external libraries. But to keep the main package slim, these libraries aren't strictly required unless you want the specific commands. As part of this change, **`ksconf --version`** now reports any issues with individual commands in the 3rd column. Any value other than 'OK' indicates a problem. Here's an example of the output if you're missing the `splunk-sdk` package.

```
...
promote         (beta)      OK
rest-export     (beta)      OK
rest-publish    (alpha)     Missing 3rd party module:  No module named splunklib.
↪client
snapshot        (alpha)     OK
...
```

Note that while the `rest-publish` command will not work in the example above, all of the other commands will continue to work fine. If you don't need `rest-publish` then there's no need to do anything about it. If you want the packages, install the "thirdparty" extras using the following command:

```
pip install ksconf[thirdparty]
```

If you want *all* the goodies:

```
pip install ksconf[fully-loaded]
```

Other issues

If you run into any issues, check out the *Validate the install* section.

3.3.7 Command line completion

Bash completion allows for a more intuitive and interactive workflow by providing quick access to command line options and file completions. Often this saves time since the user can avoid mistyping file names or be reminded of which command line actions and arguments are available without switching contexts. For example, if the user types `ksconf d` and hits `Tab`, then the `ksconf diff` is completed. Or if the user types `ksconf`, and hits `Tab` twice, the full list of command actions are listed.

This feature uses the argcomplete Python package and supports Bash, zsh, tcsh.

Install via pip:

```
pip install argcomplete
```

Enabling command line completion for ksconf can be done in two ways. The easiest option is to enable it for ksconf only. (However, it only works for the current user; it can break if the ksconf command is referenced in a non-standard way.) The alternate option is to enable global command line completion for all python scripts at once, which is preferable if you use *argparse* for many python tools.

Enable argcomplete for ksconf only:

```
# Edit your bashrc script
vim ~.bashrc

# Add the following line
eval "$(register-python-argcomplete ksconf)"

# Restart you shell, or just reload by running
source ~/.bashrc
```

To enable argcomplete globally, run the command:

```
activate-global-python-argcomplete
```

This adds a new script to your the `bash_completion.d` folder, which can be used for all scripts and all users, but it does add some minor overhead to each completion command request.

OS-specific notes:

- **Mac OS X**: The global registration option may not work as the old version of Bash was shipped by default. So either use the one-shot registration, or install a later version of bash with homebrew: `brew install bash` then. Switch to the newer bash by default with `chsh /usr/local/bin/bash`.

- **Windows**: Argcomplete doesn't work on windows Bash for GIT. See argcomplete issue 142 for more info. If you really want this, use Linux subsystem for Windows instead.

3.4 Commands

The ksconf command documentation is provided in the following ways:

1. A detailed listing of each sub-command is provided in this section. This includes relevant background descriptions, typical use cases, examples, and discussion of relevant topics. An expanded descriptions of CLI arguments and their usage is provided here. If you have not used a particular command before, start here.
2. The *Command line reference* provides a quick and convenient reference when the command line is unavailable. The same information is available by typing `ksconf <CMD> --help`. This is most helpful if you're already familiar with a command, but need a quick refresher.

Warning: Apologies for the dust

The command docs are currently undergoing reorganization. We're considering a topical layout rather than a per-command layout. Feedback and technical writing / organization contributions are highly welcomed.

Table 1: Command Listing

Command	Maturity	Description
ksconf attr-get	beta	Get the value from a specific stanzas and attribute
ksconf attr-set	beta	Set the value of a specific stanzas and attribute
ksconf check	stable	Perform basic syntax and sanity checks on .conf files
ksconf combine	beta	Combine configuration files across multiple source directories into a single destination directory. This allows for an arbitrary number of Splunk configuration layers to coexist within a single app. Useful in both ongoing merge and one-time ad-hoc use.
ksconf diff	stable	Compare settings differences between two .conf files ignoring spacing and sort order
ksconf filter	alpha	A stanza-aware GREP tool for conf files
ksconf merge	stable	Merge two or more .conf files
ksconf minimize	beta	Minimize the target file by removing entries duplicated in the default conf(s)
ksconf package	beta	Create a Splunk app .spl file from a source directory
ksconf promote	beta	Promote .conf settings between layers using either batch or interactive mode. Frequently this is used to promote conf changes made via the UI (stored in the `local` folder) to a version-controlled directory, such as `default`.
ksconf rest-export	deprecated	Export .conf settings as a curl script to apply to a Splunk instance later (via REST)
ksconf rest-publish	alpha	Publish .conf settings to a live Splunk instance via REST
ksconf snapshot	alpha	Snapshot .conf file directories into a JSON dump format
ksconf sort	stable	Sort a Splunk .conf file creating a normalized format appropriate for version control
ksconf unarchive	beta	Install or upgrade an existing app in a git-friendly and safe way
ksconf xml-format	alpha	Normalize XML view and nav files

3.4.1 ksconf

KSCONF: Ksconf Splunk CONFig tool

This utility handles a number of common Splunk app maintenance tasks in a small and easy to deploy package. Specifically, this tool deals with many of the nuances with storing Splunk apps in git and pointing live Splunk apps to a git repository. Merging changes from the live system's (local) folder to the version controlled (default) folder and dealing with more than one layer of "default" are all supported tasks which are not native to Splunk.

```
usage: ksconf [-h] [--version] [--force-color] [--disable-color]
              {attr-get,attr-set,check,combine,diff,filter,merge,minimize,package,
↪promote,rest-export,rest-publish,snapshot,sort,unarchive,xml-format}
              ...
```

Named Arguments

--version show program's version number and exit

--force-color Force TTY color mode on. Useful if piping the output a color-aware pager, like 'less -R'

--disable-color Disable TTY color mode. This can also be setup as environmental variable: `export KSCONF_TTY_COLOR=off`

3.4.2 ksconf attr-get

Get a specific stanza and attribute value from a Splunk .conf file.

```
usage: ksconf attr-get [-h] --stanza STANZA --attribute ATTR [--missing-okay]
                       [-o OUTPUT]
                       conf [conf ...]
```

Positional Arguments

conf Input file or standard input.

Named Arguments

--stanza, -s Name of the stanza within CONF to retrieve.

--attribute, --attr, -a Name of attribute within STANZA to retrieve.

--missing-okay Ignore missing stanzas and attributes.

-o, --output File where the filtered results are written. Defaults to standard out.

Example

Show the version of the Splunk AWS technology addon:

```
ksconf attr-get etc/apps/Splunk_TA_AWS/default/app.conf --stanza launcher --
↪attribute version
```

Fetch the search string for the "Internal Server Errors" search in the from my_app. The search is saved to a text file without any metadata or line continuation markers (trailing \ characters.) Note that `kconf merge` is used here to ensure that the "live" version of the search is shown, so `local` will be used if present, otherwise `default` will be shown.

```
ksconf merge $SPLUNK_HOME/etc/apps/my_app/{default,local}/savedsearches.conf \
| ksconf attr-get - -s "Internal System Errors" -a search -o errors_search.txt
```

3.4.3 ksconf attr-set

Set a specific stanza and attribute value of a Splunk .conf file. The value can be provided as a command line argument, file, or environment variable

This command does not support preserving leading or trailing whitespace. Normally this is desireable.

```
usage: ksconf attr-set [-h] --stanza STANZA --attribute ATTR
                       [--value-type TYPE] [--create-missing] [--no-overwrite]
                       conf value
```

Positional Arguments

conf Configuration file to update.

value Value to apply to the conf file. Note that this can be a raw text string, or the name of the file, or an environment variable

Named Arguments

--stanza, -s Name of the stanza within CONF to set.

--attribute, --attr, -a Name of the attribute within STANZA to set.

--value-type, -t Possible choices: string, file, env

Select the type of VALUE. The default is a string. Alternatively, the real value can be provided within a file, or an environment variable.

--create-missing Create a new conf file if it doesn't currently exist.

--no-overwrite Only set VALUE if none currently exists. This can be used to safely set a one-time default, but don't update overwrite an existing value.

Example

Update build during CI/CD

```
ksconf attr-set build/default.app -s launcher -a version 1.1.2
ksconf attr-set build/default.app -s launcher -a build --value-type env GITHUB_RUN_
↪NUMBER
```

Rewrite a saved search to match the new cooperate initiative to relabel all "CRITICAL" messages as "WHOOPSIES".

```
ksconf attr-get savedsearches.conf -s "Internal System Errors" -a search \
    | sed -re 's/CRITICAL/WHOOPSIES/g' \
    | ksconf attr-set savedsearches.conf -s "Internal System Errors" -a search --
↪value-type file -
```

Note: What if you want to write multiple stanza/attributes at once?

Of course it's possible to call `ksconf attr-set` multiple times, but that may be awkward or inefficient if many updates are needed. In the realm of shell scripting, another option is to use *ksconf merge* which is designed to merge multiple stanzas, or even multiple files, at once. With a little bit of creatively, it's possible to add (or update) and entire new stanza in-line using a single command like so:

```
printf '[drop_field(1)]\ndefinition=| fields - $field$\nargs=field\niseval=0\n' \
    | ksconf merge --in-place --target macros.conf -


# which is identical to:
ksconf merge --in-place --target macros.conf \
    <(printf '[drop_field(1)]\ndefinition=| fields - $field$\nargs=field\niseval=0\n
↪')
```

Of course, neither of these are super easy to read. If your content is static, then an easy answer it to use a static conf file. However, at some point it may be easier to just edit these using Python where any arbitrary level of complexity is possible.

Ksconf has some built in utility functions to make this kind of simple update-in-place workflow super simple. For example, the `update_conf` context manager allows access to existing conf values and quick modification. If no modification is necessary, then the file is left untouched.

```
from ksconf.conf.parser import update_conf, conf_attr_boolean

# Update app.conf for a build release
with update_conf("app.conf") as conf:
    conf["launcher"]["version"] = "1.0.2"
    conf["install"]["build"] = "33"

# Update sourcetype references in all saved searches; place marker in description
with update_conf("savedsearches.conf") as conf:
    for report in conf:
        if not conf_attr_boolean(conf[report].get("disabled", "0")):
            # Update enabled search
            search = conf[report].get("search", "")
            conf[report]["search"] = search.replace("cisco:old-understood-tech",
                                                    "cisco:new-fangled-tech")
            conf[report]["description"] = f"We did an update.\n Old description:
↪{conf[report].get('description', '')}"
```

3.4.4 ksconf check

Provides basic syntax and sanity checking for Splunk's .conf files. Use Splunk's built-in `btool check` for a more robust validation of attributes and values.

Consider using this utility as part of a pre-commit hook.

```
usage: ksconf check [-h] [--quiet] FILE [FILE ...]
```

Positional Arguments

FILE One or more configuration files to check. If '-' is given, then read a list of files to validate from standard input

Named Arguments

--quiet, -q Reduce the volume of output.

See also:

Pre-commit hooks

See *Pre-commit hooks* for more information about how the `check` command can be easily integrated in your git workflow.

How 'check' differs from btool's validation

Keep in mind that idea of *valid* in ksconf is different than within Splunk. Specifically,

- **Ksconf is more picky syntactically.** Dangling stanzas and junk lines are picked up by ksconf in general (the 'check' command or others), but silently by ignored Splunk.
- **Btool handles content validation.** The **`btool check`** mode does a great job of checking stanza names, attribute names, and values. Btool does this well and ksconf tries to not repeat things that Splunk already does well.

Why is this important?

Can you spot the error in this `props.conf`?

```
1 [myapp:web:access]
2 TIME_PREFIX = \[
3 SHOULD_LINEMERGE = false
4 category = Web
5 REPORT-access = access-extractions
6
7 [myapp:total:junk
8 TRANSFORMS-drop = drop-all
```

That's right, line 7 contains the stanza `myapp:total:junk` that doesn't have a closing `]`. How does Splunk handle this? It ignores the broken stanza header completely and therefore `TRANSFORMS-drop` gets added to the `myapp:web:access` sourcetype, which will likely result in the loss of data.

Splunk also ignores entries like this:

```
EVAL-bytes-(coalesce(bytes_in,0)+coalesce(bytes_out,0))
```

Of course here there's no `=` anywhere on the line, so Splunk just assumes it's junk and silently ignores it.

Tip: If you want to see how different this is, run ksconf check against the system default files:

```
ksconf check --quiet $SPLUNK_HOME/etc/system/default/*.conf
```

There's several files that ship with the core product that don't pass this level of validation.

Note: Key concepts

Before diving into the `combine` command, it may be helpful to brush up on the concept of *configuration layers*.

3.4.5 ksconf combine

Merge .conf settings from multiple source directories into a combined target directory. Configuration files can be stored in a `/etc/*.d` like directory structure and consolidated back into a single 'default' directory.

This command supports both one-time operations and recurring merge jobs. For example, this command can be used to combine all users' knowledge objects (stored in 'etc/users') after a server migration, or to merge a single user's settings after their account has been renamed. Recurring

operations assume some type of external scheduler is being used. A best-effort is made to only write to target files as needed.

The 'combine' command takes your logical layers of configs (upstream, corporate, Splunk admin fixes, and power user knowledge objects, ...) expressed as individual folders and merges them all back into the single `default` folder that Splunk reads from. One way to keep the 'default' folder up-to-date is using client-side git hooks.

No directory layout is mandatory, but taking advantages of the native-support for 'dir.d' layout works well for many uses cases. This idea is borrowed from the Unix System V concept where many services natively read their config files from `/etc/*.d` directories.

Version notes: dir.d was added in ksconf 0.8. Starting in 1.0 the default will switch to 'dir.d', so if you need the old behavior be sure to update your scripts.

```
usage: ksconf combine [-h] --target TARGET [-m {auto,dir.d,disable}] [-q]
                      [-I PATTERN] [-E PATTERN] [--enable-handler {jinja}]
                      [--template-vars TEMPLATE_VARS] [--dry-run]
                      [--follow-symlink] [--banner BANNER] [-K KEEP_EXISTING]
                      [--disable-marker] [--disable-cleanup]
                      source [source ...]
```

Positional Arguments

source The source directory where configuration files will be merged from. When multiple source directories are provided, start with the most general and end with the specific; later sources will override values from the earlier ones. Supports wildcards so a typical Unix `conf.d/##-NAME` directory structure works well.

Named Arguments

--target, -t Directory where the merged files will be stored. Typically either 'default' or 'local'

-m, --layer-method Possible choices: auto, dir.d, disable

Set the layer type used by SOURCE.

Use `dir.d` if you have directories like `MyApp/default.d/##-layer-name`, or use `disable` to manage layers explicitly and avoid any accidental layer detection. By default, `auto` mode will enable transparent switching between 'dir.d' and 'disable' (legacy) behavior, however this option will be removed in a future release.

-q, --quiet Make output a bit less noisy. This may change in the future...

-I, --include Name or pattern of layers to include.

-E, --exclude Name or pattern of layers to exclude from the target.

--enable-handler Possible choices: jinja

Enable optional file handling support

--template-vars Set template variables as key=value or YAML/JSON, if filename prepend with @

--dry-run, -D Enable dry-run mode. Instead of writing to TARGET, preview changes as a 'diff'. If TARGET doesn't exist, then show the merged file.

--follow-symlink, -l Follow symbolic links pointing to directories. Symlinks to files are always followed.

--banner, -b A banner or warning comment added to the top of the TARGET file. Used to discourage Splunk admins from editing an auto-generated file.

For other on-going *combine* operations, it's helpful to inform any .conf file readers or potential editors that the file is automatically generated and therefore could be overwritten again. For one-time *combine* operations, the default banner can be suppressed by passing in an empty string (`''` or `""` on Windows)

-K, --keep-existing Existing file(s) to preserve in the TARGET folder. This argument may be used multiple times.

--disable-marker Prevents the creation of or checking for the `.ksconf_controlled` marker file safety check. This file is typically used indicate that the destination folder is managed by ksconf. This option should be reserved for well-controlled batch processing scenarios.

--disable-cleanup Disable all file removal operations. Skip the cleanup phase that typically removes files in TARGET that no longer exist in SOURCE

You may have noticed similarities between the `combine` and *merge* subcommands. That's because under the covers they are using much of the same code. The combine operation essentially does a recursive merge between a set of directories. One big difference is that `combine` command will handle non-conf files intelligently, not just conf files. Additionally, `combined` can automatically detect layers for you, depending on the layering scheme in use.

Mixing layers

Just like all layers can be managed independently, they can also be combined in any way you would like. This also allows for different layers to be mixed-and-matched by selectively including layers to combine. This feature is now available in ksconf 0.8.0 and later using the `--include` and `--exclude` CLI options, which should behave as just as you'd expected.

Note: A more detailed explanation

The `--include` and `--exclude` arguments are processed in the order given. These filters are applied to all layer names. The last match wins.

If `--include` is first, then by default all layers, except for the ones explicitly included, will be excluded. Conversely, if `--exclude` is first, then all layers will be included except for the ones explicitly included. If *no* filters are given then all layers will be processed.

Here's an example, truncated for brevity, to further demonstrate how this can be used practically:

```
Splunk_TA_nix/
├── README.txt
├── bin
│   ├── bandwidth.sh
│   ├── common.sh
├── default.d
│   ├── 10-upstream
│   │   ├── app.conf
│   │   ├── data
│   │   │   └── ui
│   │   │       ├── nav
│   │   │       │   └── default.xml
│   │   │       └── views
│   │   │           └── setup.xml
│   │   ├── eventtypes.conf
│   │   ├── inputs.conf
│   │   ├── props.conf
│   │   ├── tags.conf
│   │   ├── transforms.conf
│   │   └── web.conf
│   ├── 20-common
│   │   ├── inputs.conf
│   │   ├── props.conf
│   │   └── transforms.conf
│   ├── 30-master-apps
│   │   └── inputs.conf
│   └── 30-shcluster-apps
│       ├── inputs.conf
│       └── web.conf
├── lookups
│   ├── nix_da_update_status.csv
│   ├── nix_da_version_ranges.csv
└── metadata
    └── default.meta
```

Here we have several named layers in play:

- `10-upstream` - the layer used to contain the default app content that ships from the Splunk TA, or whatever is "upstream" source is.
- `20-common` - organizational level change to deployed everywhere.
- `30-master-apps` - The bits that should just go to the indexers.

- `30-shcluster-apps` - Content that should go to just the search heads.

In this case, we always want to combine the `10-*` and `20-*` layers, but only want to include either the master or searchhead cluster layer depending on server role.

```
ksconf combine src/Splunk_TA_nix --target build/shcd/Splunk_TA_nix \
    --exclude=30-* --include=30-shcluster-apps
ksconf combine src/Splunk_TA_nix --target build/cm/Splunk_TA_nix \
    --exclude=30-* --include=30-master-apps

# Say you just want the original app, for some reason:
ksconf combine src/Splunk_TA_nix --target /build/orig/Splunk_TA_nix --include=10-
↪upstream
```

Using this technique you can pretty quickly write some simple shell scripts to build these all at once:

```
for role in shcluster master
do
    ksconf combine src/Splunk_TA_nix \
        --target build/${role}/Splunk_TA_nix \
        --exclude=30-* --include=30-${role}-apps
done
```

Hopefully this gives you some ideas on how you can start to build some custom workflows with just a few small shell scripts.

Layer methods

Ksconf supports different methods of layer detection mechanism. Right now just two different schemes are supported, but if you have other ways of organizing your layers, please *reach out*.

Directory.d (`dir.d`)
: Also known as `*.d` directory layout is allows layers to be embedded on a directory structure that allows for simple prioritization and labels to be applied to each layer. Anyone who's configured a Linux server should find this familiar.

  Example: `MyApp/default.d/10-my_layer/props.conf`

  Convention: `<directory-name>.d/<##>-<layer-name>/`

  When these layers are combined, the top level folder is modified to remove the trailing `.d`, and all content from the enable layers is combined within that folder. The layer-name portion of the path is discarded in the final combined path. Content is combined based on the assigned ranking of each layer, or directory sort order.

Disable (legacy)
: If you would prefer to stick with the previous behavior (no automatic detection of layers) and specify all *SOURCE* directories manually, then use this mode. In this

mode, each layer must be explicitly defined (or provide as a wildcard) and any other files operations must be handled elsewhere.

Auto (default)
In auto mode, if more than one source directory is given, then `disable` mode is used, if only a single directory is given then `dir.d` will be used.

How do I pick?

Mode	Useful when	Avoid if
`dir.d`	• Building a full app • If you need layers in multiple places (`default.d`, and `lookups.d`) • If you sometimes have no layers, then combine falls back to a file copy	• Have existing `.d` folders with other meaning • Have multiple source directories.
`disable`	• Highly customized work flows / full-control over combination logic	• For app build scripts.

Examples

Merging a multilayer app

Let's assume you have a directory structure that looks like the following. This example features the Cisco Security Suite.

```
Splunk_CiscoSecuritySuite/
├── README
├── default.d
│   ├── 10-upstream
│   │   ├── app.conf
│   │   ├── data
│   │   │   └── ui
│   │   │       ├── nav
│   │   │       │   └── default.xml
│   │   │       └── views
│   │   │           ├── authentication_metrics.xml
│   │   │           ├── cisco_security_overview.xml
│   │   │           ├── getting_started.xml
│   │   │           ├── search_ip_profile.xml
│   │   │           ├── upgrading.xml
│   │   │           └── user_tracking.xml
```

(continues on next page)

(continued from previous page)

```
    │   ├── eventtypes.conf
    │   ├── macros.conf
    │   ├── savedsearches.conf
    │   └── transforms.conf
    ├── 20-my-org
    │   └── savedsearches.conf
    ├── 50-splunk-admin
    │   ├── indexes.conf
    │   ├── macros.conf
    │   └── transforms.conf
    └── 70-firewall-admins
        ├── data
        │   └── ui
        │       └── views
        │           ├── attacks_noc_bigscreen.xml
        │           ├── device_health.xml
        │           └── user_tracking.xml
        └── eventtypes.conf
├── lookups
├── metadata
└── static
```

In this structure, you can see several layers of configurations at play:

1. The `10-upstream` layer appears to be the version of the default folder that shipped with the Cisco app.
2. The `20-my-org` layer is small and only contains tweaks to a few saved search entries.
3. The `50-splunk-admin` layer represents local settings changes to specify index configurations, and to augment the macros and transformations that ship with the default app.
4. And finally, `70-firewall-admins` contains some additional view (2 new, and 1 existing). Note that since `user_tracking.xml` is not a `.conf` file it will fully replace the upstream default version (that is, the file in `10-upstream`)

You can merge all these layers inside this app into a new app folder using the command below:

```
ksconf combine repo/Splunk_CiscoSecuritySuite --target=shcluster/apps/Splunk_
↪CiscoSecuritySuite
```

`ksconf` will automatically detect the `default.d` folder as a layer-containing directory and merge content from the detected layers (`10-upstream`, `20-my-org`, `...`) into a new `default` folder in the resulting app. All other content (such as *README, bin*, *static*, *lookups* and so on) will be copied as-is.

Changed in version 0.8: If you are using `ksconf` before 0.8, then you have to manually merge the layers, and possibly copy other top-level folders on your own (outside of ksconf). The example below still works fine after version 0.8, but the default behavior may change in 1.0, so it's advisable to start using `--layer-method` explicitly in any scripts you may use.

Here are the commands that could be used to generate a new (merged) `default` folder from all of the layers shown above.

```
cd Splunk_CiscoSecuritySuite
ksconf combine default.d/* --target=default
```

Note that in the example above, the `default` folder now lives along side the `default.d` folder. Also note that *only* the contents of `default.d` are copied, not the entire app, like in the above example.

See also:

The *unarchive* command can be used to install or upgrade apps stored in a version controlled system in a layer-aware manor.

Consolidating 'users' directories

The `combine` command can consolidate 'users' directory across several instances after a phased server migration. See *Migrating the 'users' folder*.

3.4.6 ksconf diff

Compares the content differences of two .conf files

This command ignores textual differences (like order, spacing, and comments) and focuses strictly on comparing stanzas, keys, and values. Note that spaces within any given value, will be compared. Multi-line fields are compared in a more traditional 'diff' output so that long saved searches and macros can be compared more easily.

```
usage: ksconf diff [-h] [-o FILE] [--detail {global,stanza,key}] [--comments]
                   [--format {diff,json}]
                   CONF1 CONF2
```

Positional Arguments

CONF1 Left side of the comparison

CONF2 Right side of the comparison

Named Arguments

-o, --output	File where difference is stored. Defaults to standard out.
--detail, -d	Possible choices: global, stanza, key Control the highest level for which 'replace' events may occur.
--comments, -C	Enable comparison of comments. (Unlikely to work consistently)
--format, -f	Possible choices: diff, json Output file format to produce. 'diff' the the classic format used by default. 'json' is helpful when trying to review changes programmatically.

Example

Add screenshot here

To use `ksconf diff` as an external diff tool, check out *Ksconf as external difftool*.

3.4.7 ksconf filter

Filter the contents of a conf file in various ways. Stanzas can be included or excluded based on a provided filter or based on the presence or value of a key.

Where possible, this command supports GREP-like arguments to bring a familiar feel.

```
usage: ksconf filter [-h] [-o FILE] [--comments] [--verbose] [--skip-broken]
                     [--match {regex,wildcard,string}] [--ignore-case]
                     [--invert-match] [--files-with-matches]
                     [--count | --brief] [--stanza PATTERN]
                     [--attr-present ATTR] [--attr-matches ATTR PATTERN]
                     [--attr-not-matches ATTR PATTERN] [-e | -d]
                     [--keep-attrs WC-ATTR] [--reject-attrs WC-ATTR]
                     CONF [CONF ...]
```

Positional Arguments

CONF	Input conf file

Named Arguments

-o, --output File where the filtered results are written. Defaults to standard out.

--comments, -C Preserve comments. Comments are discarded by default.

--verbose Enable additional output.

--skip-broken Skip broken input files. Without this things like duplicate stanzas and invalid entries will cause processing to stop.

--match, -m Possible choices: regex, wildcard, string

Specify pattern matching mode. Defaults to 'wildcard' allowing for * and ? matching. Use 'regex' for more power but watch out for shell escaping. Use 'string' to enable literal matching.

--ignore-case, -i Ignore case when comparing or matching strings. By default matches are case-sensitive.

--invert-match, -v Invert match results. This can be used to show what content does NOT match, or make a backup copy of excluded content.

Output mode

Select an alternate output mode. If any of the following options are used, the stanza output is not shown.

--files-with-matches, -l List files that match the given search criteria

--count, -c Count matching stanzas

--brief, -b List name of matching stanzas

Stanza selection

Include or exclude entire stanzas using these filter options.

All filter options can be provided multiple times. If you have a long list of filters, they can be saved in a file and referenced using the special `file://` prefix. One entry per line. Entries can be either a literal strings, wildcards, or regexes, depending on MATCH.

--stanza Match any stanza who's name matches the given pattern. PATTERN supports bulk patterns via the `file://` prefix.

--attr-present Match any stanza that includes the ATTR attribute. ATTR supports bulk attribute patterns via the `file://` prefix.

--attr-matches, --attr-eq Match any stanza containing ATTR == PATTERN. PATTERN supports the special `file://filename` syntax. Matching can be a direct string comparison (equals), or a regex and wildcard match.

Note that all `--attr-match` and `--attr-not-match` arguments are matched together. For a stanza to match, all rules must apply. If

attr is missing from a stanza, the value becomes an empty string for matching purposes.

--attr-not-matches, --attr-ne Match any stanza containing ATTR != PATTERN. See `--attr-matches` for additional details.

-e, --enabled-only Keep only enabled stanzas. Any stanza containing `disabled = 1` will be removed. The value of `disabled` is assumed to be false by default.

-d, --disabled-only Keep disabled stanzas only. The value of the *disabled* attribute is interpreted as a boolean.

Attribute selection

Include or exclude attributes passed through. By default, all attributes are preserved. Allowlist (keep) operations are preformed before blocklist (reject) operations.

--keep-attrs Select which attribute(s) will be preserved. This space separated list of attributes indicates what to preserve. Supports wildcards.

--reject-attrs Select which attribute(s) will be discarded. This space separated list of attributes indicates what to discard. Supports wildcards.

How is this different that btool?

Some of the things filter can do functionally overlaps with **`btool list`**. Take for example:

```
ksconf filter search/default/savedsearches.conf --stanza "Messages by minute last 3␣
↪hours"
```

Is essentially the same as:

```
splunk btool --app=search savedsearches list "Messages by minute last 3 hours"
```

The output is the same, assuming that you didn't overwrite any part of that search in `local`. But if you take off the `--app` argument, you'll quickly see that `btool` is merging all the layers together to show the final value of all attributes. That is certainly a helpful thing to do, but not always what you want.

Ksconf is *only* going to look at the file you explicitly pointed it to. It doesn't traverse the tree on it's own. This means that it works on app directory structure that live inside or outside of your Splunk instance. If you've ever tried to run `btool check` on an app that you haven't installed yet, then you'll understand the value of this.

In many other cases, the usage of both `ksconf filter` and `btool` differ significantly.

Note: What if I want a filter default & local at the same time?

In situations where it would be beneficial to filter based on the combined view of default and local, then simply use *ksconf_cmd_merge* first. Here are two options.

Option 1: Use a named temporary file

```
ksconf merge search/{default,local}/savedsearches.conf > savedsearches.conf
ksconf filter savedsearches.conf - --stanza "* last 3 hours"
```

Option 2: Chain both commands together

```
ksconf merge search/{default,local}/savedsearches.conf | ksconf filter --stanza "*␣
↪last 3 hours"
```

Examples

Searching for attribute/values combinations

Find all enabled input stanzas with a sourcetype prefixed with `apache:`.

```
ksconf filter etc/apps/*/{default,local}/inputs.conf \
    --enabled-only --attr-eq sourcetype 'apache:*'
```

List the names of saved searches using potentially expensive search commands:

```
ksconf filter etc/apps/*/{default,local}/savedsearches.conf \
    -b --match regex \
    --attr-eq search '.*\|\s*(streamstats|transaction) .*'
```

Show sourcetype stanzas where `EVENT_BREAKER` is defined but not enabled:

```
ksconf filter etc/deployment-apps/*/{default,local}/props.conf \
    --skip-broken --match regex \
    --attr-match-equals EVENT_BREAKER '.+' \
    --attr-match-not-equals EVENT_BREAKER_ENABLE '(true|1)'
```

Note that both conditions listed must match for a stanza to match. Logical 'AND' not an 'OR'. Also note the use of `--skip-broken` because sometimes Splunk base apps have invalid conf files.

Lift and shift

Copy all indexes defined within a specific app.

```
cd $SPLUNK_DB
for idx in $(ksconf filter $SPLUNK_HOME/etc/app/MyApp/default/indexes.conf --brief)
do
    echo "Copy index ${idx}"
    tar -czf "/migrate/export-${idx}" "${idx}"
done
```

Now you'll have a copy all of the necessary indexes in the `/migrate` folder to make *MyApp* work on another Splunk instance. Of course, there's likely other migration tasks to consider, like copying the actual app. This is just one way ksconf can help.

Can I do the same thing with standard unix tools?

Sure, go for it!

Yes, there's significant overlap with the filter command and what you can do with **grep**, **awk**, or **sed**. Much of that is on purpose, and in fact some command line arguments were borrowed.

I used to do these tasks by hand, but it's easy to make mistakes. The idea of **ksconf** is to give you stable and reliable tools that are more suitable for `.conf` file work. Also keep in mind that these features are expanding much more quickly than the unix tools change.

Although, if you've had to deal with BSD vs GNU tools and trying to find a set of common arguments, then you probably already appreciate how awesome a domain-specific-tool like this is.

3.4.8 ksconf merge

Merge two or more .conf files into a single combined .conf file. This is similar to the way that Splunk logically combines the `default` and `local` folders at runtime.

```
usage: ksconf merge [-h] [--target TARGET] [--ignore-missing] [--in-place]
                    [--dry-run] [--banner BANNER]
                    conf [conf ...]
```

Positional Arguments

conf The source configuration file(s) to collect settings from.

Named Arguments

--target, -t Destination file for merged configurations. If not provided, the merged conf is written to standard output.

--ignore-missing, -s Silently ignore any missing CONF files.

--in-place, -i Enable in-place update mode. When selected, the TARGET file will also be considered as the base of the merge operation. All CONF files will be merged with TARGET. When disabled, any existing content within TARGET is ignored and overwritten.

The `--in-place` option was added in v0.12.1. In earlier version of ksconf, and moving forward, this same behavior can be accomplished by simply listing the target twice. Once as in the `--target` option, and then a second time as the first CONF file.

--dry-run, -D Enable dry-run mode. Instead of writing to TARGET, preview changes in 'diff' format. If TARGET doesn't exist, then show the merged file.

--banner, -b A banner or warning comment added to the top of the TARGET file. Used to discourage Splunk admins from editing an auto-generated file.

Examples

Here is an elementary example that merges all `props.conf` file from *all* of your technology addons into a single output file:

```
ksconf merge --target=all-ta-props.conf etc/apps/*TA*/{default,local}/props.conf
```

See an expanded version of this example here: *Building an all-in one TA for your indexing tier*

3.4.9 ksconf minimize

See also:

See the *Minimizing files* for background on why this is important.

Minimize a conf file by removing any duplicated default settings.

Reduce a local conf file to only your intended changes without manually tracking which entries you've edited. Minimizing local conf files makes your local customizations easier to read and often results in cleaner upgrades.

```
usage: ksconf minimize [-h] [--target TARGET] [--dry-run | --output OUTPUT]
                       [--explode-default] [-k PRESERVE_KEY]
                       CONF [CONF ...]
```

Positional Arguments

CONF The default configuration file(s) used to determine what base settings are. The base settings determine what is unnecessary to repeat in target file.

Named Arguments

--target, -t The local file that you wish to remove duplicate settings from. This file will be read from and then replaced with a minimized version.

--dry-run, -D Enable dry-run mode. Instead of writing and minimizing the TARGET file, preview what would be removed as a 'diff'.

--output Write the minimized output to a separate file instead of updating TARGET.

This option can be used to *preview* the actual changes. Sometimes if `--dry-run` mode produces too much output, it's helpful to look at the actual minimized version of the file in concrete form (rather than a relative format, like a diff.) This may also be helpful in other workflows.

--explode-default, -E Enable minimization across stanzas for special use-cases. Helpful when dealing with stanzas downloaded from a REST endpoint or `btool list` output.

This mode will not only minimize the same stanza across multiple config files, it will also attempt to minimize any default values stored in the `[default]` or global stanza as well. For this to be effective, it's often necessary to include system-level defaults in the CONF list. For example, to trim out cruft in savedsearches.conf, make sure you add `etc/system/default/savedsearches.conf` as an input.

-k, --preserve-key Specify attributes that should always be kept.

Example usage

```
cd Splunk_TA_nix
cp default/inputs.conf local/inputs.conf

# Edit 'disabled' and 'interval' settings in-place
vi local/inputs.conf

# Remove all the extra (unmodified) bits
ksconf minimize --target=local/inputs.conf default/inputs.conf
```

Undoing a minimize

You can use *ksconf merge* to reverse the effect of minimize by running a command like so:

```
ksconf merge default/inputs.conf local/inputs.conf
```

Additional capabilities

For special cases, the `--explode-default` mode reduces duplication between entries in normal stanzas (as normal) and then additionally reduces duplication between individual stanzas and default entries. Typically you only need this mode if your dealing with a conf file that's been fully expanded to include all the layers, which doesn't happen under normal circumstances. This does happen anytime you download a stanza from a REST endpoint or munged together output from `btool list`. If you've ever done this with `savedsearches.conf` stanzas, you'll be painfully aware of how massive they are! This is the exact use case that `--explode-default` was written for.

In such a case, it may be helpful to minimize against the full definition of *default*, which effectively requires looking at all the layers of default. This includes all global app settings, and system-level settings.

There are limitations to this approach.

- You have to manually list out all the layers. (Sometimes just pointing to the system-level defaults is good enough)
- Minimize doesn't take namespace into account. This means ownership, sharing, and ACLs are ignored.

In many ways `minimize` mimics what Splunk does *every* time it updates a conf file, as discussed in *How Splunk writes to conf files*. If you find yourself frequently needing the power of `--explode-default`, at some point a potentially better approach may be to simply post stanzas to the REST endpoint. However, this typically does a good enough job, especially for offline scenarios.

Additionally, this command doesn't strictly require a bloated file. For example, if `disabled = 0` is both a global default, and set on a per-stanza basis, that could be reduced too. However, typically this isn't super helpful.

3.4.10 ksconf package

Create a Splunk app or add on tarball (`.spl`) file from an app directory.

`ksconf package` can do useful things like, exclude unwanted files, combine layers, set the application version and build number, drop or promote the `local` directory into `default`.

Note that some arguments, like the `FILE` support special values that can be automatically evaluated at runtime. For example the placeholders `{{version}}` or `{{git_tag}}` can be expanded into the output tarball filename.

If both layering and templating are in use at the same time, be aware that templates are rendered prior to layering operations. This allows, for example, one layer to include a simple `indexes.conf` file and another layer to include an `indexes.conf.j2` template.

```
usage: ksconf package [-h] [-f SPL] [--app-name APP_NAME]
                      [--blocklist BLOCKLIST] [--allowlist ALLOWLIST]
                      [--layer-method {dir.d,disable}] [-I PATTERN]
                      [-E PATTERN] [--enable-handler {jinja}]
                      [--template-vars TEMPLATE_VARS] [--follow-symlink]
                      [--set-version VERSION] [--set-build BUILD]
                      [--allow-local | --block-local | --merge-local]
                      [--release-file RELEASE_FILE]
                      SOURCE
```

Positional Arguments

SOURCE Source directory for the Splunk app.

Named Arguments

-f, --file Name of splunk app file (tarball) to create. Placeholder variables in `{{var}}` syntax can be used here.

--app-name Specify the top-level app folder name. If this is not given, the app folder name is automatically extracted from the basename of SOURCE. Placeholder variables, such as `{{app_id}}` can be used here.

--blocklist, -b Pattern for files/directories to exclude. Can be given multiple times. You can load multiple exclusions from disk by using `file://path` which can be used with `.gitignore` for example. (Default includes: `.git*`, `*.py[co]`, `__pycache__`, `.DS_Store`)

--allowlist, -a Remove a pattern that was previously added to the blocklist.

--enable-handler Possible choices: jinja

Enable optional file handling support

--template-vars Set template variables as key=value or YAML/JSON, if filename prepend with @

--follow-symlink, -l Follow symbolic links pointing to directories. Symlinks to files are always followed.

--set-version Set application version. By default the application version is read from default/app.conf. Placeholder variables such as `{{git_tag}}` can be used here.

--set-build Set application build number.

--allow-local Allow the `local` folder to be kept as-is WARNING: This goes against Splunk packaging practices, and will cause AppInspect to fail. However, this option can be useful for private package transfers between servers, app backups, or other admin-like tasks.

--block-local Block the `local` folder and `local.meta` from the package.

--merge-local Merge any files in `local` into the `default` folder during packaging. This is the default behavior.

Layer filtering

If the app being packaged includes multiple layers, these arguments can be used to control which ones should be included in the final app file. If no layer options are specified, then all layers will be included.

--layer-method Possible choices: dir.d, disable

Set the layer type used by SOURCE. Additional description provided in in the `combine` command.

-I, --include Name or pattern of layers to include.

-E, --exclude Name or pattern of layers to exclude from the target.

Advanced Build Options

The following options are for more advanced app building workflows.

--release-file Write the path of the newly generated archive file (SPL) after the archive is written. This is useful in build scripts when the SPL contains variables so the final name may not be known ahead of time.

Variables

The following variables are currently available for use during package building. These are referenced using the `{{var}}` syntax. See the implementation in `AppVarMagic` if you'd like to contribute additional variables.

Supported Variables

Variable	Sour	Notes
app_id	app.c	Get id from [package] in app.conf. This must be the app folder name for any app published to Splunkbase.
build	app.c	Get build from [install] in app.conf
versio	app.c	Get version from [launcher] in app.conf
git_ta	git	Run git describe --tags --always --dirty. Common prefixes are removed such as v or release- from the tag name.
git_la	git	Run git log -n1 --pretty=format:%h -- .
git_he	git	Run git rev-parse --short HEAD
layers_	layers	List of unique ksconf layers used to build the app. Layers are separated by an double underscores (__). If no layers were used then an empty string is returned.
layers_	layers	Unique hash of unique ksconf layers used. This is a truncated SHA256 of the layers_list variable.

Example

```
ksconf package -f my_app.tgz MyApp
```

A more realistic example where the version number in app.conf is managed by some external process, possibly a tool like bumpversion.

```
bumpversion minor
ksconf package MyApp \
    --set-version={{git_tag}} \
    -f dist/my_app-{{version}}.tgz \
    --release-file=.artifact
echo "Build complete, upload $(<.artifact) to SplunkBase"
```

This will output a message like: `Build complete, upload dist/my_app-1.3.0.tgz to SplunkBase`

And of course this workflow could be further automated using Splunkbase API calls.

See also

More sophisticated builds can be achieved using the `BuildManager`

3.4.11 ksconf promote

Propagate .conf settings applied in one file to another. Typically this is used to move `local` changes (made via the UI) into another layer, such as the `default` or a named `default.d/50-xxxxx`) folder.

Promote has two modes: batch and interactive. In batch mode, all changes are applied automatically and the (now empty) source file is removed. In interactive mode, the user is prompted to select stanzas to promote. This way local changes can be held without being promoted.

NOTE: Changes are *MOVED* not copied, unless `--keep` is used.

```
usage: ksconf promote [-h] [--batch | --interactive | --summary | --diff]
                      [--verbose] [--match {regex,wildcard,string}]
                      [--ignore-case] [--invert-match] [--stanza PATTERN]
                      [--force] [--keep] [--keep-empty]
                      SOURCE TARGET
```

Positional Arguments

SOURCE The source configuration file to pull changes from. (Typically the `local` conf file)

TARGET Configuration file or directory to push the changes into. (Typically the `default` folder)

Named Arguments

--batch, -b Use batch mode where all configuration settings are automatically promoted. All changes are removed from source and applied to target. The source file will be removed unless `--keep-empty` is used.

--interactive, -i Enable interactive mode where the user will be prompted to approve the promotion of specific stanzas and attributes. The user will be able to apply, skip, or edit the changes being promoted.

--summary, -s Summarize content that could be promoted.

--diff, -d Show the diff of what would be promoted.

--verbose Enable additional output.

--force, -f Disable safety checks. Don't check to see if SOURCE and TARGET share the same basename.

--keep, -k Keep conf settings in the source file. All changes will be copied into the TARGET file instead of being moved there. This is typically a bad idea since local always overrides default.

--keep-empty Keep the source file, even if after the settings promotions the file has no content. By default, SOURCE will be removed after all content has been moved into TARGET. Splunk will re-create any necessary local files on the fly.

Automatic filtering options

Include or exclude stanzas to promote using these filter options. Stanzas selected by these filters will be promoted.

All filter options can be provided multiple times. If you have a long list of filters, they can be saved in a file and referenced using the special `file://` prefix. One entry per line.

--match, -m Possible choices: regex, wildcard, string

Specify pattern matching mode. Defaults to 'wildcard' allowing for `*` and `?` matching. Use 'regex' for more power but watch out for shell escaping. Use 'string' to enable literal matching.

--ignore-case Ignore case when comparing or matching strings. By default matches are case-sensitive.

--invert-match, -v Invert match results. This can be used to prevent content from being promoted.

--stanza Promote any stanza with a name matching the given pattern. PATTERN supports bulk patterns via the `file://` prefix.

> **Warning:** The promote command **moves** configuration settings between *SOURCE* and *TARGET* and therefore both files are updated. This is unlike most other commands where only *TARGET* is modified. Using the `--keep` argument will prevent *SOURCE* from being updated.

Modes

Promote has different modes:

Batch mode
Changes are applied automatically and the (now empty) source file is removed by default. The source file can be retained by using either the `--keep` or `--keep-empty` arguments, see descriptions above.

Interactive mode
Prompts the user to pick which stanzas and attributes to integrate. In practice, it's common that not all local changes will be ready to be promoted and committed at the same time.

Hint: This mode was inspired by **`git add --patch`** command.

Summary mode
Shows the user a brief breakdown of what stanzas are available for promotion. This can be used to simply the use of the `--stanza` filtering options (automatic promotion) to show the names of stanzas available for promotion. Note that when `--summary` and `--stanza` are used at the same time, then the summary output will include any output not *already* matched by `--stanza` filter.

Default
If you haven't specified either batch or interactive mode, you'll be asked to pick one at startup. You'll be given the option to show a diff, apply all changes, or be prompted to keep or reject changes interactively.

Automated promotions

Ksconf 0.7.8 added support for automatic stanza matching and promotion using a `ksconf filter`-like CLI options.

Key features include:

Automatic promotion of stanzas
One or more named stanzas can be promoted automatically using the `--stanza` argument. This argument can be given multiple times to match multiples stanzas at once. In batch mode, only the named stanzas will be promoted; but in interactive mode, the named stanzas will be promoted first, and any content remaining to be promoted can be handled interactively.

Matching mode
Like with the `ksconf filter` command, multiple methods of matching are supported. This includes: string matching (default), wildcard (or "glob") matching, and regular expressions.

Inversion
The `--invert-match` option allows for the selection to be inverted. In this mode, it's possible to select which stanzas should *not* be promoted. This can be used as a blocklist to prevent accidental promotions.

Safety checks

Moving content between files is a potentially risky operation. Here are some of the safety mechanisms that ksconf has in place to prevent data loss.

Tip: Pairing ksconf with a version control tool like **`git`**, while not required, does provide another layer of protection against loss or corruption. If you promote and commit changes frequently, then the scope of potential loss is reduced.

Syntax checking
Strong syntax checking is enabled for both *SOURCE* and *TARGET* to prevent mistakes, such as dangling or duplicate stanzas, which could lead to even more corruption.

File fingerprinting
Various attributes of the *SOURCE* and *TARGET* files are captured at startup and compared again before any changes are written to disk. This reduces the possibility of a race-condition on a live Splunk system. This mostly impacts interactive mode because the session lasts longer. If this is a concern, run promote only when Splunk is offline.

Same file check
Attempts to promote content from a file to itself are prevented. While logically no one would want to do this, in practice having a clear error message saves time and confusion.

Base name check
The *SOURCE* and *TARGET* should share the same base name. In other words, trying to promote from `inputs.conf` into `props.conf` (due to a typo) will be prevented. This matters more in batch mode. In interactive mode, it should be pretty obvious that the type of entries don't make sense and therefore the user can simply exit without saving.

For scripting purposes, there may be times where pushing changes between arbitrary-named files is helpful, so this check can be bypassed by using the `--force` argument.

Note: Unfortunately, the unit testing coverage for the `promote` command is quite low. This is primarily because I haven't yet figured out how to handle unit testing for interactive CLI tools (as this is the only interactive command to date.) I'm also not sure how much the UI may change; Any assistance in this area would be greatly appreciated.

Examples

A simple promotion looks like this.

```
ksconf promote local/props.conf default/props.conf
```

This is equivalent to this minor shortcut.

```
ksconf promote local/props.conf default
```

In this case, ksconf determines that `default` is a directory and therefore assumes that you want the same filename, `props.conf` in this case.

Tip: Using a directory as TARGET may seem like a trivial improvement, but in practice it greatly

reduces accidental cross-promotion of content. Therefore, we suggest its use.

Similarly, a shortcut for pushing between metadata files exists:

```
ksconf promote metadata/local.meta metadata
```

A few example of automatic promotion of a named stanza:

```
# Single stanzas
ksconf promote local/savedsearches.conf default --stanza "My fancy search"

# Wildcard promote all prod server alerts
ksconf promote local/savedsearches.conf default --match wildcard --stanza
↪"Server PRD* Alert"

# Automatically promote everything except for one search:
ksconf promote local/savedsearches.conf default --batch --invert-match --
↪stanza "Local test"
```

Interactive mode

Keyboard shortcuts

Key	Meaning	Description
y	Yes	Apply changes
n	No	Don't apply
d	Diff	Show the difference between the file or stanza.
q	Quit	Exit program. Don't save changes.

Limitations

- Currently, an attribute-level section has not be implemented. Entire stanzas are either kept local or promoted fully.
- Interactive mode currently lacks "help". In the meantime, see the keyboard shortcuts listed above.
- At present, comments in the *SOURCE* file will not be preserved.
- If *SOURCE* or *TARGET* is modified externally while promote is running, the entire operation will be aborted, thus loosing any custom selections you made in interactive mode. This needs improvement.
- There's currently no way to preserve certain local settings with some kind of "never-promote" flag. It's not uncommon to have some settings in `inputs.conf`, for example, that you never want to promote.

- There is no *dry-run* mode supported. Primarily, this is because it would only work for batch mode, and in interactive mode you explicitly see exactly what will be changed before anything is applied. (If you really need a dry-run for batch mode, use *ksconf merge* to show the result of *TARGET SOURCE* combined.)

3.4.12 ksconf rest-export

Deprecated since version 0.7.0: You should consider using *ksconf rest-publish* instead of this one. The only remaining valid use case for `rest-export` (this command) is for disconnected scenarios. In other words, if you need to push stanzas to a Splunkd instance where you don't (and can't) install `ksconf`, then this command may still be useful to you. In this case, `ksconf rest-export` can create a shell script that you can transfer to the correct network, and then run the shell script. But for **ALL** other use cases, the `rest-publish` command is superior.

Build an executable script of the stanzas in a configuration file that can be later applied to a running Splunk instance via the Splunkd REST endpoint.

This can be helpful when pushing complex props and transforms to an instance where you only have UI access and can't directly publish an app.

```
usage: ksconf rest-export [-h] [--output FILE] [--disable-auth-output]
                          [--pretty-print] [-u | -D] [--url URL] [--app APP]
                          [--user USER] [--owner OWNER] [--conf TYPE]
                          [--extra-args EXTRA_ARGS]
                          CONF [CONF ...]
```

Positional Arguments

CONF Configuration file(s) to export settings from.

Named Arguments

--output, -t Save the shell script output to this file. If not provided, the output is written to standard output.

-u, --update Assume that the REST entities already exist. By default, output assumes stanzas are being created.

-D, --delete Remove existing REST entities. This is a destructive operation. In this mode, stanza attributes are unnecessary and ignored. NOTE: This works for 'local' entities only; the default folder cannot be updated.

--url URL of Splunkd. Default: "https://localhost:8089"

--app Set the namespace (app name) for the endpoint

--user Deprecated. Use –owner instead.

--owner Set the object owner. Typically, the default of 'nobody' is ideal if you want to share the configurations at the app-level.

--conf Explicitly set the configuration file type. By default, this is derived from CONF, but sometimes it's helpful to set this explicitly. Can be any valid Splunk conf file type. Examples include: 'app', 'props', 'tags', 'savedsearches', etc.

--extra-args Extra arguments to pass to all CURL commands. Quote arguments on the command line to prevent confusion between arguments to ksconf vs curl.

Output Control

--disable-auth-output Turn off sample login curl commands from the output.

--pretty-print, -p Enable pretty-printing. Make shell output a bit more readable by splitting entries across lines.

> **Warning:** For interactive use only!
>
> This command is indented for manual admin workflows. It's quite possible that shell escaping bugs exist that may allow full shell access if you put this into an automated workflow. Evaluate the risks, review the code, run as a least-privilege user, and be responsible.

Roadmap

For now, the assumption is that `curl` command will be used. (Patches to support the Power Shell `Invoke-WebRequest` cmdlet would be greatly welcomed!)

Example

```
ksconf rest-export --output=apply_props.sh etc/app/Splunk_TA_aws/local/props.conf
```

3.4.13 ksconf rest-publish

Note: This command effectively replaces *ksconf rest-export* for nearly all use cases. The only thing that `rest-publish` can't do that `rest-export` can, is handle a disconnected scenario. But for **ALL** other use cases, the `rest-publish` (this command) command is far superior.

Note: This commands requires the Splunk Python SDK, which is automatically bundled with the *Splunk app for KSCONF*.

Publish stanzas in a .conf file to a running Splunk instance via REST. This requires access to the HTTPS endpoint of Splunk. By default, ksconf will handle both the creation of new stanzas and the update of existing stanzas.

This can be used to push full configuration stanzas where you only have REST access and can't directly publish an app.

Only attributes present in the conf file are pushed. While this may seem obvious, this fact can have profound implications in certain situations, like when using this command for continuous updates. This means that it's possible for the source .conf to ultimately differ from what ends up on the server's .conf file. One way to avoid this, is to explicitly remove an object using `--delete` mode first, and then insert a new copy of the object. Of course, this means that the object will be unavailable. The other impact is that diffs only compares and shows a subset of attribute.

Be aware, that for consistency, the configs/conf-TYPE endpoint is used for this command. Therefore, a reload may be required for the server to use the published config settings.

```
usage: ksconf rest-publish [-h] [--conf TYPE] [-m META] [--url URL]
                           [--user USER] [--pass PASSWORD] [-k]
                           [--session-key SESSION_KEY] [--app APP]
                           [--owner OWNER] [--sharing {user,app,global}] [-D]
                           CONF [CONF ...]
```

Positional Arguments

CONF Configuration file(s) to export settings from.

Named Arguments

--conf Explicitly set the configuration file type. By default, this is derived from CONF, but sometimes it's helpful to set this explicitly. Can be any valid Splunk conf file type. Examples include: 'app', 'props', 'tags', 'savedsearches', etc.

-m, --meta Specify one or more `.meta` files to determine the desired read & write ACLs, owner, and sharing for objects in the CONF file.

--url URL of Splunkd. Default: "https://localhost:8089"

--user Login username Splunkd. Default: "admin"

--pass Login password Splunkd. Default: "changeme"

-k, --insecure Disable SSL cert validation.

--session-key	Use an existing session token instead of using a username and password to login.
--app	Set the namespace (app name) for the endpoint
--owner	Set the user who owns the content. The default of 'nobody' works well for app-level sharing.
--sharing	Possible choices: user, app, global Set the sharing mode.
-D, --delete	Remove existing REST entities. This is a destructive operation. In this mode, stanza attributes are unnecessary. NOTE: This works for 'local' entities only; the default folder cannot be updated.

Examples

A simple example:

```
ksconf rest-publish etc/app/Splunk_TA_aws/local/props.conf \
    --user admin --password secret --app Splunk_TA_aws --owner nobody --sharing␣
↪global
```

This command also supports replaying metdata like ACLs:

```
ksconf rest-publish etc/app/Splunk_TA_aws/local/props.conf \
    --meta etc/app/Splunk_TA_aws/metdata/local.meta \
    --user admin --password secret --app Splunk_TA_aws
```

3.4.14 ksconf snapshot

Build a static snapshot of various configuration files stored within a structured json export format. If the .conf files being captured are within a standard Splunk directory structure, then certain metadata and namespace information is assumed based on typical path locations. Individual apps or conf files can be collected as well, but less metadata may be extracted.

```
usage: ksconf snapshot [-h] [--output FILE] [--minimize] PATH [PATH ...]
```

Positional Arguments

PATH Directory from which to load configuration files. All .conf and .meta file are included recursively.

Named Arguments

--output, -o Save the snapshot to the named files. If not provided, the snapshot is written to standard output.

--minimize Reduce the size of the JSON output by removing whitespace. Reduces readability.

> **Warning: Output NOT stable!**
>
> The output from this command hasn't really been tested in any kind of serious way for usability. Consider this a proof-of-concept. Anyone interested in this type of functionality should *reach out* to discuss uses cases.

Example

```
ksconf snapshot --output=daily-$(date +%Y-%m-%d).json $SPLUNK_HOME/etc/app/
```

3.4.15 ksconf sort

Sort a Splunk .conf file. Sort has two modes: (1) by default, the sorted config file will be echoed to the screen. (2) the config files are updated in-place when the `-i` option is used.

Manually managed conf files can be protected against changes by adding a comment containing the string `KSCONF-NO-SORT` to the top of any .conf file.

```
usage: ksconf sort [-h] [--target FILE | --inplace] [-F] [-q] [-n LINES]
                   FILE [FILE ...]
```

Positional Arguments

FILE Input file to sort, or standard input.

Named Arguments

--target, -t File to write results to. Defaults to standard output.

--inplace, -i Replace the input file with a sorted version.

WARNING: This a potentially destructive operation that may move/remove comments.

-n, --newlines Number of lines between stanzas.

In-place update arguments

-F, --force Force file sorting for all files, even for files containing the special 'KSCONF-NO-SORT' marker.

-q, --quiet Reduce the output. Reports only updated or invalid files. This is useful for pre-commit hooks, for example.

See also:

Pre-commit hooks

See *Pre-commit hooks* for more information about how the `sort` command can be easily integrated in your git workflow.

Examples

To recursively sort all files

```
find . -name '*.conf' | xargs ksconf sort -i
```

3.4.16 ksconf unarchive

Install or overwrite an existing app in a git-friendly way. If the app already exists, steps will be taken to upgrade it safely.

The `default` folder can be redirected to another path (i.e., `default.d/10-upstream` or other desirable path if you're using the `ksconf combine` tool to manage extra layers).

```
usage: ksconf unarchive [-h] [--dest DIR] [--app-name NAME]
                        [--default-dir DIR] [--exclude EXCLUDE] [--keep KEEP]
                        [--allow-local]
                        [--git-sanity-check {off,changed,untracked,ignored}]
                        [--git-mode {nochange,stage,commit}] [--no-edit]
                        [--git-commit-args GIT_COMMIT_ARGS]
                        SPL
```

Positional Arguments

SPL The path to the archive to install.

Supports tarballs (.tar.gz, .spl), and less-common zip files (.zip)

Named Arguments

--dest Set the destination path where the archive will be extracted. By default, the current directory is used. Sane values include: etc/apps, etc/deployment-apps, and so on.

Often this will be a git repository working tree where Splunk apps are stored.

--app-name The app name to use when expanding the archive. By default, the app name is taken from the archive as the top-level path included in the archive (by convention).

Expanding archives that contain multiple (ITSI) or nested apps (NIX, ES) is not supported.

--default-dir Name of the directory where the default contents will be stored. This is a useful feature for apps that use a dynamic default directory that's created and managed by the 'combine' mode.

--exclude, -e Add a file pattern to exclude from extraction. Splunk's pseudo-glob patterns are supported here. `*` for any non-directory match, `...` for ANY (including directories), and `?` for a single character.

--keep, -k Specify a pattern for files to preserve during an upgrade. Repeat this argument to keep multiple patterns.

--allow-local Allow local/* and local.meta files to be extracted from the archive.

Shipping local files is a Splunk app packaging violation so local files are blocked to prevent customizations from being overridden.

--git-sanity-check By default, `git status` is run on the destination folder to detect working tree or index modifications before the unarchive process starts, but this is configurable. Sanity check choices go from least restrictive to most thorough:

- Use `off` to prevent any 'git status' safety checks.
- Use `changed` to abort only upon local modifications to files tracked by git.
- Use `untracked` (the default) to look for changed and untracked files before considering the tree clean.
- Use `ignored` to enable the most intense safety check which will abort if local changes, untracked, or ignored files are found.

--git-mode Possible choices: nochange, stage, commit

Set the desired level of git integration. The default mode is *stage*, where new, updated, or removed files are automatically handled for you.

To prevent any `git add` or `git rm` commands from being run, pick the 'nochange' mode.

If a git commit is incorrect, simply roll it back with `git reset` or fix it with a `git commit --amend` before the changes are pushed anywhere else. There's no native `--dry-run` or undo for unarchive mode because that's why you're using git in the first place, right? (Plus, such features would require significant overhead and unit testing.)

--no-edit Tell git to skip opening your editor on commit. By default, you will be prompted to review/edit the commit message. (Git Tip: Delete the content of the default message to abort the commit.)

--git-commit-args, -G Extra arguments to pass to 'git'

Note: What if I'm not using version control?

Sanity checks and commit modes are automatically disabled if the app is being installed into a directory that is *not* contained within a git working tree. Ksconf confirms that *git* is present and functional before running sanity checks.

3.4.17 ksconf xml-format

Normalize and apply consistent XML indentation and CDATA usage for XML dashboards and navigation files.

Technically this could be used on *any* XML file, but certain element names specific to Splunk's simple XML dashboards are handled specially, and therefore could result in unusable results.

The expected indentation level is guessed based on the first element indentation, but can be explicitly set if not detectable.

```
usage: ksconf xml-format [-h] [--indent INDENT] [--quiet] FILE [FILE ...]
```

Positional Arguments

FILE	One or more XML files to check. If '-' is given, then a list of files is read from standard input

Named Arguments

--indent	Number of spaces. This is only used if indentation cannot be guessed from the existing file.
--quiet, -q	Reduce the volume of output.

See also:

Pre-commit hooks

See *Pre-commit hooks* for more information about how the `xml-format` command can be integrated in your git workflow.

NOTE: While it may work on other XML files, it hasn't been tested for other files, and therefore is not recommended as a general-purpose XML formatter. Specific awareness of various Simple XML tags is baked into this product.

Note: This command requires the external `lxml` Python module.

This package was specifically selected (over the built-in 'xml.etree' interface) because it (1) supports round-trip preservation of CDATA blocks, and (2) already ships with Splunk's embedded Python.

This is an optional requirement, unless you want to use the `xml-format` command.

As of v0.12.0, this is not longer installed by the `ksconf` package. However, if you are using pre-commit hooks from the ksconf-pre-commit repo for the `ksconf-xml-format` hook.

Why is this important?

TODO: Note the value of using `<!CDATA[[ ]]>` blocks.

Value of consistent indentation.

To recursively format xml files

```
find . -path '*/data/ui/views/*.xml' -o -path '*/data/ui/nav/*.xml' | ksconf xml-
↪format -
```

3.5 Cheat Sheet

Here's a quick rundown of handy `ksconf` commands:

Note: Note that for clarity, most of the command line arguments are given in their long form.

Long commands may be broken across line for readability. When this happens, a trailing backslash (\) is shown. This can be copied verbatim into many shells.

Contents

3.5.1 General purpose

Extracting a single value

Grabbing the definition of a single macro using *ksconf attr-get*. Note in the case of a complex or multi-line expression, any line continuation characters will be removed.

```
ksconf attr-get macros.conf --stanza 'unroll_json_array(6)' --attribute␣
↪definition
```

Updating a single value

Suppose you have a macro called `mydata_index` that defines the source indexes for your dashboards. The following command uses *ksconf attr-set* to update that macro directly from the CLI without opening an editor.

```
ksconf attr-set macros.conf --stanza mydata_index --attribute definition --
↪value 'index=mydata1 OR index=otheridx'
```

In this case the definition is a single line, but multi-line input is handled automatically. It's also possible to pull a vale from an existing file or from an environment variable, should that be useful.

Comparing files

Show the differences between two conf files using *ksconf diff*.

```
ksconf diff savedsearches.conf savedsearches-mine.conf
```

Sorting content

Create a normalized version of a configuration file, making conf files easier to merge with **`git`**. Run an in-place sort like so:

```
ksconf sort --inplace savedsearches.conf
```

Tip: Use the `ksconf-sort` *pre-commit* hook to do this for you.

Extract specific stanza

Say you want to *grep* your conf file for a specific stanza pattern:

```
ksconf filter search/default/savedsearches.conf --stanza 'Errors in the␣
↪last *'
```

Say you want to list stanzas containing `cron_schedule`:

```
ksconf filter Splunk_TA_aws/default/savedsearches.conf --brief \
    --attr-present 'cron_schedule'
```

Remove unwanted settings

Say you want to remove `vsid` from a legacy savedsearches file:

```
ksconf filter search/default/savedsearches.conf --reject-attrs "vsid"
```

To see just to the scheduled time and enablement status of scheduled searches, run:

```
ksconf filter Splunk_TA_aws/default/savedsearches.conf \
    --attr-present cron_schedule \
    --keep-attrs 'cron*' \
    --keep-attrs enableSched
    --keep-attrs disabled
```

List apps configured in the deployment server

```
ksconf filter -b serverclass.conf --stanza 'serverClass:*:app:*' | \
    cut -d: -f4 | sort | uniq
```

Find saved searches with earliest=-1d@d

```
ksconf filter apps/*/default/savedsearches.conf \
    --attr-eq dispatch.earliest_time "-1d@d"
```

3.5.2 Cleaning up

Reduce cruft in local

If you're in the habit of copying the *default* files to *local* in the TAs you deploy, here is a quick way to 'minimize' your files. This will reduce the *local* file by removing all the *default* settings you copied but didn't change. (The importance of this is outlined in *Minimizing files*.)

```
ksconf minimize Splunk_TA_nix/default/inputs.conf --target Splunk_TA_nix/
↪local/inputs.conf
```

Pushing local changes to default

App developers can push changes from the `local` folder to the `default` folder:

```
ksconf promote --interactive myapp/local/props.conf myapp/default/props.
↪conf
```

You will be prompted to pick which items you want to promote. Alternatively, use the `--batch` option to promote everything in one step, without reviewing the changes first.

3.5.3 Packaging and building apps

Quick package and install

Use the `--release-file` option of the package command to write out the name of the final created tarball. This helps when the final tarball name isn't known in advance because it contains a version string, for example. By simply placing the latest release in a static location, this allows commonly repeated operations, like build+install to be chained together in a convenient way making iterations quite fast.

```
cd my-apps
ksconf package --release-file .release kintyre_app_speedtest &&
    "$SPLUNK_HOME/bin/splunk" install app "$(<.release)" -update 1
```

A build process for the same package, where the version is defined by the latest git tag, would look something like this:

```
ksconf package -f "dist/kintyre_app_speedtest-{{version}}.tar.gz" \
    --set-version="{{git_tag}}" \
    --set-build=$GITHUB_RUN_NUMBER \
    --release-file .release \
    kintyre_app_speedtest
echo "Go upload $(<.release) to Splunkbase"
```

3.5.4 Advanced usage

Migrating content between apps

Say you want to move a bunch of savedsearches from `search` into a more appropriate app. First create a file that lists all the names of your searches (one per line) in `corp_searches.txt`. Next, copy just the desired stanzas, to your new `corp_app` application using the following command:

```
ksconf filter --match string --stanza 'file://corp_searches.txt' \
    search/local/savedsearches.conf --output corp_app/default/
↪savedsearches.conf
```

Because we want to *move*, not just *copy*, the searches, they can now be removed from the search app using the following steps:

```
ksconf filter --match string --stanza 'file://corp_searches.txt' \
    --invert-match search/local/savedsearches.conf \
    --output search/local/savedsearches.conf.NEW

# Backup the original
mv search/local/savedsearches.conf \
    /my/backup/location/search-savedsearches-$(date +%Y%M%D).conf

# Move the updated file in place
mv search/local/savedsearches.conf.NEW search/local/savedsearches.conf
```

Note: Setting the matching mode to `string` prevents any special characters that may be present in your search names from being interpreted as wildcards.

Migrating the 'users' folder

Say you stood up a new Splunk server and the migration took longer than expected. Now you have two `users` folders and don't want to loose all the goodies stored in either one. You've copied the users folder to `user_old`. You're working from the new server and would generally prefer to keep whatever is on the new server over what is on the old. (This is because some of your users copied over some of their critical alerts manually while waiting for the migration to complete, and they've made updates they don't want to lose.)

After stopping Splunk on the new server, run the following commands.

```
mv /some/share/users_old  $SPLUNK_HOME/etc/users.old
mv $SPLUNK_HOME/etc/users $SPLUNK_HOME/etc/users.new

ksconf combine $SPLUNK_HOME/etc/users.old $SPLUNK_HOME/etc/users.new \
    --target $SPLUNK_HOME/etc/users --banner ''
```

Now double check the results and start Splunk.

Using `--banner` essentially disables the output banner feature. Because, in this case, the combine operation is a one-time job and therefore no top-of-file warning is needed.

Maintaining apps stored in a local git repository

Extract and commit a new/updated app

```
ksconf unarchive --git-mode=commit my-package-112.tgz
```

For apps that use layers (`default.d` folder), then use a command like so:

```
ksconf unarchive --git-mode=commit \
    --default-dir=default.d/10-upstream \
    --keep 'default.d/*' my-package-112.tgz
```

If you'd like to disable git hooks, like pre-commit, when importing a new release of an upsteam app, add `--git-commit-args="--no-verify` to the above commands.

3.5.5 Putting it all together

Pulling out a stanza defined in both default and local

Say you wanted to count the number of searches containing the word `error`

```
ksconf merge default/savedsearches.conf local/savedsearches.conf \
    | ksconf filter - --stanza '*Error*' --ignore-case --count
```

This is a simple example of chaining two basic **ksconf** commands together to perform a more complex operation. The first command handles the merge of default and local `savedsearches.conf` into a single output stream. The second command filters the resulting stream finding stanzas containing the word 'Error'.

Building an all-in one TA for your indexing tier

Say you need to build a single TA containing all the index-time settings for your indexing tier. (Note: Enterprise Security does something similar when generating the indexer app.)

```
ksconf merge etc/apps/*TA*/{default,local}/props.conf \
    | ksconf filter - --output=TA-for-indexers/default/props.conf \
      --include-attr 'TRANSFORMS*' \
      --include-attr 'TIME_*' \
      --include-attr 'MUST_BREAK*' \
      --include-attr 'SHOULD_LINEMERGE' \
      --include-attr 'EVENT_BREAKER*' \
      --include-attr 'LINE_BREAKER*'
```

This example is incomplete because it doesn't list *every* index-time `props.conf` attribute, and leaves out `transforms.conf` and `fields.conf`, but hopefully you get the idea.

3.6 Plugins

Ksconf supports a growing number of plugins to enable custom workflow and and elegantly handle custom use cases that don't make sense to implement in the core tool. Plugins functionality is implemented using pluggy.

Note that, much like the pluggy docs themselves, we use the term "hook" and "plugin" are used interchangeably at times. Generally, the term "hook" is a specific handoff point where control can be passed from the ksconf codebase to some hook function that you've implemented to perform a specific operation. The term "plugin" refers to a package (or collection) of implemented hooks.

There are multiple ways of enabling these hooks or collections, but the easiest way is by means of registration process built into Python's packaging system. This means that by simply installing a package, brand new functionality can be enabled within your `ksconf` command line. Over time, we hope that more of these plugins can be published and made available to a wider audience on pypi.

3.6.1 Using plugins

Existing plugins can be found on pypi by search for the ksconf-* package prefix. With a little bit of Python experience, it's relatively simple to write your own.

Installation should be as simple as using your favorite package manager to install the plugin. For example:

```
pip install ksconf-<plugin-name>
```

Once installed, you can confirm which plugins are loaded and activated using `--version`.

```
ksconf --version
```

Output:

```
_                   ___
| |_ ___ ___ ___ ___|  _|
| '_|_ -|  _| . |   |  _|
|_,_|___|___|___|_|_|_|

ksconf 0.11.6.dev3+e508597.dirty
Python: 3.9.16  (/Users/username/venv/bin/python)
Git SHA1 e508597d committed on 2023-09-20
Installed at: /Users/username/sandbox/ksconf
Platform:  Darwin Kernel Version 22.6.0: Wed Jul  5 22:22:05 PDT 2023; root:xnu-8796.
↪141.3~6/RELEASE_ARM64_T6000
Git support:  (/usr/bin/git) git version 2.39.2 (Apple Git-143)
```

(continues on next page)

(continued from previous page)

```
Plugins:
  package ksconf-jinja-markdown (1.0.0) from /Users/lalleman/ksconf/plugins/jinja-
↪markdown/ksconf_jinja_markdown.py
    hook    modify_jinja_env via add_jinja_filters
...
```

Note that your installation will likely look different.

3.6.2 Troubleshooting

Review hook execution

Currently enabling hook monitoring is handled by `KSCONF_DEBUG` which also controls several other troubleshooting operations, such as enabling stack traces when exceptions occur.

Disable individual plugins

Plugins can be temporarily banned by using the `KSCONF_PLUGIN_DISABLE` environment variable.

```
# Block for your entire session (or add to ~/.bashrc?)
export KSCONF_PLUGIN_DISABLE="jinja-markdown test-plugin2"

# Quick interactive ban (for a quick test)
KSCONF_PLUGIN_DISABLE=jinja-markdown ksconf package ...
```

To permanently ban the plugin, simply remove the corresponding python package.

```
pip uninstall ksconf-jinja-markdown
```

3.6.3 List of plugins

All plugins are defined within `KsconfHookSpecs`.

3.6.4 Plugin examples

Modify Jinja Environment

The `modify_jinja_env()` hook allows for modification of the Jinja2 environment so that custom filters can be added. This very specific hook allows a rendered Jinja2 layer file to use custom Jinja filter, so that in this case, markdown content can be rendered as HTML.

```
from ksconf.hook import ksconf_hook
from jinja2 import Environment

def markdown_to_html(md):
    """ Jinja filter for markdown to html """
    import commonmark
    return commonmark.commonmark(md)

@ksconf_hook(specname="modify_jinja_env")
def add_jinja_filters(env: Environment):
    """ Register new filter(s) to the Jinja environment, for use within templates. ""
↪"
    env.filters["markdown2html"] = markdown_to_html
```

This specific example is bundled up as python package and is installable via:

```
pip install ksconf-jinja-markdown
```

3.6.5 Packaging a Plugin

Packing is fairy easy, and there are examples in the `plugins` folder in the ksconf GitHub repository. This example assumes your packing a plugin that lives in a `ksconf/plugins/fancy_plugin.py`. Note that the `ksconf/plugins` is a top-level directory that puts your new plugin in the `ksconf.plugins` namespace. (This isn't technically required, but it's the recommended approach.)

Here's an example of a `setup.py` file:

```
from setuptools import setup, find_namespace_packages

setup(name="ksconf-fancy-plugin",
      version="0.5.0",
      install_requires=[
          "ksconf>=0.13.0",
          "some-fancy-library",   # Add 3rd party libraries here, if needed
      ],
      entry_points={"ksconf_plugin": ["fancy-plugin = ksconf.plugins.fancy_plugin"]},
      packages=find_namespace_packages(include=['ksconf.plugins.*']),
      py_modules=["ksconf_fancy_plugin"],
      description="Adds general fanciness within Ksconf",
      classifiers=["Environment :: Plugins"],
      author="Your name",
      author_email="your@name.example",
      url="Repo",
      zip_safe=False)
```

Then simply build and install your package.

```
pip install .
```

If you need to remove it, you can always run:

```
pip uninstall ksconf-fancy-plugin
```

All python package building and general development best practices apply, but this should be enough to get you started.

3.7 Contributing

Pull requests are greatly welcome! If you plan on contributing code back to the main `ksconf` repo, please follow the standard GitHub fork and pull-request work-flow. We also ask that you enable a set of git hooks to help safeguard against avoidable issues.

3.7.1 Pre-commit hook

The ksconf project uses the pre-commit hook to enable the following checks:

- Fixes trailing whitespace, EOF, and EOLs
- Confirms python code compiles (AST)
- Blocks the committing of large files and keys
- Rebuilds the dynamic portions of the docs related to the CLI.
- Confirms that all unit tests pass. (Currently, this is the same test run by Travis CI, but since tests complete in under 5 seconds, the run-everywhere approach seems appropriate for now. Eventually, the local testing will likely become a subset of the full test suite.)

Note: Multiple uses of pre-commit

Be aware, that the ksconf repo uses pre-commit for validation of it's own content, and ksconf-pre-commit repo provides a pre-commit hook service definition for other repos. The first scenario is discussed in this section of the guide. The second scenario is for repositories that house Splunk apps to use *ksconf check* and *ksconf sort* as easy to use hooks against their own `.conf` files which is discussed further in *Pre-commit hooks*.

Installing the pre-commit hook

To ensure your changes comply with the ksconf coding standards, please install and activate pre-commit.

Install:

```
pip install pre-commit

# Register the pre-commit hooks (one time setup)
cd ksconf
pre-commit install --install-hooks
```

Install gitlint

Gitlint will check to ensure that commit messages are in compliance with the standard subject, empty-line, and body format. You can enable it with:

```
gitlint install-hook
```

3.7.2 Refresh module listing

After making changes to the module hierarchy or simply adding new commands, refresh the listing for the autodoc extension by running the following command. Note that this may not remove old packages.

```
sphinx-apidoc --force -o "docs/source/api" ksconf 'ksconf/ext'
```

3.7.3 Create a new subcommand

Checklist:

1. Create a new module in `ksconf.commands.<CMD>`.
   - Create a new class derived from `KsconfCmd`. You must, at a minimum, define the following methods:
     - `register_args()` to setup any config parser inputs.
     - `run()` which handles the actual execution of the command.
2. Register a new entrypoint configuration in the `setup_entrypoints.py` script. Edit the `_entry_points` dictionary to add an entry for the new command.
   - Each entry must include command name, module, and implementation class.
3. Create unit tests in `test/test_cli_<CMD>.py`.

4. Create documentation in `docs/source/cmd_<CMD>.rst`. You'll want to build the docs locally to make sure everything looks correct. Part of the documentation is automatically generated from the argparse arguments defined in the `register_args()` method, but other bits need to be spelled out explicitly.

When in doubt, it may be helpful to look back over history in git for other recently added commands and use that as an example.

Here's an overview of paths you should expect to update:

File path	Description / purpose
`ksconf/commands/fancy.py`	The core python code and CLI interface
`tests/tests/test_cli_CMD.py`	Add new unit test here
`docs/source/cmd_CMD.rst`	Command line documentation. Make sure to include the *argparse* module
`ksconf/setup_entrypoints.py`	Add a new entrypoint line here, or the new command won't be registered
`.pre-commit-hooks.yaml`	If a new command is applicable, add this to the ksconf-pre-commit repo.
`requirements.txt`	Update if there are any new external dependencies
`make_splunk_app`	If there's new dependencies that need to go into the Splunk app

3.7.4 Cookiecutter options

The following example assume we're make a new command called `asciiart`:

```
git clone https://github.com/Kintyre/ksconf.git
cd ksconf

# Kick off a cookiecutter  (promt submodule: asciiart)
cookiecutter https://github.com/Kintyre/ksconf.git -c cookiecutter-subcommand

cp ksconf-asciiart/* .

git add ksconf/commands/*.py docs/source/cmd_*.rst tests/test_cli*.py

# Merge that one line into entrypoints
vim ksconf/setup_entrypoints*.py
git add kconf/setup_entrypoints.py

# Now run pre-commit to ensure that the new command is found successfully and is␣
↪importable
```

(continues on next page)

(continued from previous page)

```
pre-commit
# Now go write code, tests, docs and commit ...
```

3.8 Developer setup

The following steps highlight the developer install process.

3.8.1 Tools

If you are a developer, then we strongly suggest installing into a virtual environment to prevent overwriting the production version of ksconf and for the installation of the developer tools. (The virtual environment name `venv` is used below, but this can be whatever suites, just make sure not to commit it.)

```
git clone https://github.com/Kintyre/ksconf.git
cd ksconf

# Setup and activate virtual environment
python3 -m venv venv
. venv/bin/activate

# Install developer packages
pip install -r requirements-dev.txt

# Install the ksconf package in '--editable' mode
pip install -e .
```

3.8.2 Install ksconf

```
git clone https://github.com/Kintyre/ksconf.git
cd ksconf
pip install .
```

3.8.3 Building the docs

```
cd ksconf
. venv/bin/activate

cd docs
make html
open build/html/index.html
```

If you are actively editing the docs, and would like changes to be updated in your browser as you save changes `.rst` files, then use the script in the root directory:

```
./make_docs
```

If you'd like to build PDF, then you'll need some extra tools. On Mac, you may also want to install the following (for building docs, etc.):

```
brew install homebrew/cask/mactex-no-gui
```

3.8.4 Running TOX

Local testing across multiple versions of python can be accomplished with tox and pyenv. See the online docs for theses tools for more details.

Tox and pyenv can be run like so:

```
# Install the necessary python versions
pyenv install 3.7.9
...
pyenv install 3.10.1

# Set specific default version of python for each major/minor release (tab␣
↪completion is your friend here)
pyenv local 3.7.9 ... 3.10.1

# Run tox for ALL python versions
tox

# Run tox for just a specific python version
tox -e py38
```

Some additional information about how to setup and run these tests can be gleaned from the `Vagrantfile` and `Dockerfile` in the root of the git repository, though specific python versions contained there may be quite out of date.

3.9 Git tips & tricks

These tips & tricks are based on prior Splunk, git, and ksconf experience. None of this content is an endorsement of a particular approach or tool. Read the docs, and take responsibility. As always, your millage may vary.

3.9.1 Pre-commit hooks

Ksconf is setup to work as a pre-commit plugin. To use ksconf in this manner, simply configure the ksconf repo in your pre-commit configuration file. If you haven't done any of this before, it's not difficult to setup but is beyond the scope of this guide. We suggest that you read the pre-commit docs and review this section when you are ready to setup the hooks.

Hooks provided by ksconf

Three hooks are currently defined by the *ksconf-pre-commit repo*:

ksconf-check
: Runs *ksconf check* to perform basic validation tests against all files in your repo that end with `.conf` or `.meta`. Any errors will be reported by the UI at commit time and you'll be able to correct mistakes before bogus files are committed into your repo. If you're not sure why you'd need this, check out *Why validate my conf files?*

ksconf-sort
: Runs *ksconf sort* to normalize any of your `.conf` or `.meta` files which will make diffs more readable and merging more predictable. As with any hook, you can customize the filename pattern of which files this applies to. For example, to manually organize `props.conf` files, simply add the `exclude` setting. *See Example below.*

ksconf-xml-format:
: Runs *ksconf xml-format* to apply consistency to your XML representations of Simple XML dashboards and navigation files. Dashboard Studio views can also be formatted too, along with the nested JSON payload. Formatting includes appropriate indention and the automatic addition of `<![CDATA[ ... ]]>` blocks, as needed, to reduce the need for XML escaping, resulting in more readable source file. By default, this hook looks at standard locations where XML views and navigation typically live.

Repository Change

As of October 2023 (v0.12), the ksconf pre-commit hooks have been moved into their own repository to simplify packing and dependency complexities. This will impact users whenever upgrading their pre-commit configs to use the latest version of ksconf. This will happen, for example, when running `pre-commit autoupdate`.

To be clear, this change will not break any existing pre-commit configuration. But to avoid any disruption, we suggest you start using this new repository now, while you're thinking about it. The change is easy.

Migration Steps

Edit your `.pre-commit-config.yaml` file to (1) use the new `repo` location, and (2) use a recent version in `rev` (v0.11.7+)

Replace this:

```
- repo: https://github.com/Kintyre/ksconf
  rev: v0.9.5
```

with this:

```
- repo: https://github.com/Kintyre/ksconf-pre-commit
  rev: v0.11.9
```

Alternately, you could run the following shell commands:

```
# Update pre-commit config in-place
sed -e 's~https://github.com/Kintyre/ksconf$~https://github.com/Kintyre/ksconf-pre-
↪commit~' -i.bak .pre-commit-config.yaml

# Update to latest release
pre-commit autoupdate --repo https://github.com/Kintyre/ksconf-pre-commit
```

Configuring pre-commit hooks in you repo

To add ksconf pre-commit hooks to your repository, add the following content to your `.pre-commit-config.yaml` file:

```
repos:
- repo: https://github.com/Kintyre/ksconf-pre-commit
  rev: v0.11.9
  hooks:
    - id: ksconf-check
    - id: ksconf-sort
    - id: ksconf-xml-format
```

For general reference, here's a copy of what we frequently use for our repos.

```
- repo: https://github.com/pre-commit/pre-commit-hooks
  rev: v2.0.0
  hooks:
    - id: trailing-whitespace
      exclude: README.md
    - id: end-of-file-fixer
      exclude: README.md$
    - id: check-json
```

(continues on next page)

(continued from previous page)

```
    - id: check-xml
    - id: check-ast
    - id: check-added-large-files
      args: [ '--maxkb=50' ]
    - id: check-merge-conflict
    - id: detect-private-key
    - id: mixed-line-ending
      args: [ '--fix=lf' ]
- repo: https://github.com/Kintyre/ksconf-pre-commit
  rev: v0.11.9
  hooks:
    - id: ksconf-check
    - id: ksconf-sort
      exclude: (props|logging)\.conf
    - id: ksconf-xml-format
```

Tip: You should update `rev` to the most currently released stable version. Upgrading this frequently isn't typically necessary since these two operations are pretty basic and stable. However, it's still a good idea to review the change log to see what, if any, pre-commit functionality was updated.

Note: Sometimes pre-commit can get in the way.

Instead of disabling it entirely, it's often better to disable the specific rule that's causing an issue using the `SKIP` environmental variable. So for example, if intentionally adding a file over 50 Kb, a command like this will allow all the *other* rules to still run.

```
SKIP=check-added-large-file git commit -m "Refresh lookup files for bogus TA"
```

This and other tricks are fully documented in the pre-commit docs. However, this comes up frequently enough that it's worth repeating here.

Should my version of ksconf and pre-commit plugins be the same?

If you're running both `ksconf` locally as well as the ksconf pre-commit plugin, then technically you have ksconf installed twice. That may sound less than ideal, but practically, this isn't a problem. As long as the version of the ksconf CLI tool is *close* to the `rev` listed in `.pre-commit-config.yaml`, then everything should work fine.

Our suggestion:

1. Keep versions in the same *major.minor* release range or bump the version every 6-12 months.
2. Check the changelog for any pre-commit related changes or compatibility concerns.

While keeping `ksconf` CLI versions in sync across your environment is recommended, it doesn't matter as much for the pre-commit plugin. Why?

1. The pre-commit plugin offers a small subset of overall ksconf functionality.
2. The exposed functionality is stable and changes infrequently.
3. Updating pre-commit too frequently may cause unnecessary delays if you have a large team or high number of git clones throughout your environment, as each one will have to wait and upgrade the next time pre-commit is kicked off.

3.9.2 Git configuration tweaks

Ksconf as external difftool

Use *ksconf diff* as an external *difftool* provider for **`git`**. Edit `~/.gitconfig` and add the following entries:

```
[difftool "ksconf"]
    cmd = "ksconf --force-color diff \"$LOCAL\" \"$REMOTE\" | less -R"
[difftool]
    prompt = false
[alias]
    ksdiff = "difftool --tool=ksconf"
```

Now you can run this new `git` alias to compare files in your directory using the `ksconf diff` feature instead of the default textual diff that git provides. This is especially helpful if the `ksconf-sort` pre-commit hook hasn't been enabled.

```
git ksdiff props.conf
```

Tip: Wonky version of git?

If you find yourself in the situation where `git-difftool` hasn't been fully installed correctly (or the Perl extensions are missing), then here's a workaround option for you.

```
ksconf diff <(git show HEAD:./props.conf) props.conf
```

Take note of the relative path prefix `./`. In practice, this can be problematic.

Stanza aware textual diffs

Make `git diff` show the 'stanza' on the `@@` output lines.

Note: How does git know that?

Ever wonder how `git diff` is able to show you the name of the function or method where changes were made? This works for many programming languages out of the box. If you've ever spent much time looking at diffs, that additional context is invaluable. As it turns out, this is customizable by adding a stanza matching regular expression with a file pattern match.

Simply add the following settings to your git configuration:

```
[diff "conf"]
    xfuncname = "^(\\[.*\\])$"
```

Then register this new ability with specific file patterns using git's `attributes` feature. Edit `~/.config/git/attributes` and add:

```
*.conf diff=conf
*.meta diff=conf
```

Note: Didn't work as expected?

Be aware that the location for your global-level attributes may be different. Use the following command to test if the settings have been applied.

```
git check-attr -a -- *.conf
```

Test to make sure the `xfuncname` attribute was set as expected:

```
git config diff.conf.xfuncname
```

3.9.3 Git tricks

Avoid replicating the .git folder

Version controlling certain directories, like `master-apps` or `shcluster` can result in the entire `.git` folder being replicated to other Splunk instances. This can be problematic because (1) this folder can be quite large, and (2) it can cause confusion on the receiving side leaving an admin to believe that the destination folder is version controlled. Splunk doesn't provide a way to block the `.git` folder from being replicated.

Generally, there may be other more appropriate way to control content of these folders, but when faced with this situation, a simple workaround is to move the real `.git` folder to a secondary location (outside of the replicated folder) and instead us a `.git` file with a `gitdir:` pointer to the

real git folder. This is may sound complicated, but it's quite easy in practice. Here's an example with a `master-apps` folder:

```
cd $SPLUNK_HOME/etc/master-apps
mv -v "${PWD}/.git" "${PWD}.git"
echo "gitdir: ${PWD}.git" > "$PWD/.git"
```

After running the above commands, the `.git` folder is now named `master-apps.git`, and `master-apps/.git` is now just a small file referencing the new location of the git repository folder. Splunk deployment/synchronization operations now just copy a small file, rather than the `.git` folder.

More information is available at gitrepository-layout.

3.10 Random

3.10.1 Typographic and Convention

Pronounced: k·s·knf

Capitalization:

Form	Acceptability factor
`ksconf`	Always lower for CLI. Generally preferred.
KSCONF	Okay for titles.
Ksconf	Title case is okay too.
KSConf	You'll see this, but weird.
KSconf	Just proper nouns capitalized
KsConf	No, except maybe in a class name?
KsconF	Thought about it. No go! Reserved for ASCII art ONLY

> I wrote this while laughing at my own lack of consistency.
> – Lowell

3.10.2 How Splunk writes to conf files

Splunk does some counter intuitive thing when it writes to local conf files.

For example,

1. All conf file updates are automatically minimized. Splunk never has to write the entire contents because updates *only* happen to "local" files.
2. Modified stanzas are sometimes rewritten in place, and other times removed from the current position and moved to the bottom of the .conf file. This behavior appears to vary based on what REST endpoint is used to initiate the update.

3. New stanzas are written with attributes sorted lexicographically. When a stanza is updated in place, the modified attributes may be updated in place and new entires are typically added at the bottom of the stanza.
4. Sometimes boolean values persist in unexpected ways. Primarily this is because there's more than one way to represent them textually, and that textual representation is different than what's stored in default. Often, literal values are passed through a conf REST POST so they make it to disk, but when read, are translated into booleans.

Essentially, Splunk will always "minimize" the conf file at each update. This is because Splunk internally keeps track of the final representation of the entire stanza (in memory), and only when it's written to disk does Splunk care about the current contents of the local file. In fact, Splunk re-reads the conf file immediately before updating it. This is why, if you've made a local changes and forgot to reload, Splunk will typically not lose your changes. (Unless you've updated the same attribute both places... I mean, it's not magic.)

Tip: Don't believe me? Try it yourself.

To prove that it works this way, simply find a saved search that you modified from any app that you installed. Look at the local conf file and observe your changes. Now, go edit the saved search and restore some attribute to it's original value; the most obvious one here would be the `search` attribute, but that's tricky if it's multiple lines. Now, go look at the local conf file again. If you've updated it with *exactly* the same value, then that attribute will have been completely removed from the local file. This is in fact a neat trick that can be used to revert local changes to allow future updates to "pass-though" unimpeded. In SHC scenarios, this may be your only option to remove local settings.

Okay, so what's the value in having a *minimize* command if Splunk does this automatically every time it's makes a change? Well, simply put, because Splunk can't write to all local file locations. Splunk only writes to the local folders of system, etc/users, and etc/apps (and sometimes to deployment-apps app.conf local file, but that's a different topic).

Also, there are times where boolean values will show up in an unexpected manor because of how Splunk treats them internally. It isn't certain if this is a silly mistake in the default .conf files or a clever workaround to what's essentially a design flaw in the conf system. Either way, we suspect the user benefits. Because Splunk accepts more values as boolean than what it will write out, certain boolean values will always be explicitly stored in the conf files. This means that `disabled` and several other settings in `savedsearches.conf` always get explicitly written. How is that helpful? Well, imagine what would happen if you accidentally changed `disabled = 1` in the global stanzas in savedsearches.conf. Well, *nothing* if all savedsearches have that values explicitly written. The point is this: there are times when repeating yourself isn't a bad thing. (Incidentally, this is the reason for the `--preserve-key` flag on the *minimize* command.)

3.10.3 Grandfather Paradox

The KSCONF Splunk app disadvantageously breaks it's designed paradigm. Ksconf was designed to be the program that manages all your other apps, so by deploying ksconf as an app itself, we open up the possibility that ksconf could upgrade, deploy, or manage itself. Basically, it could cut off the limb that it's standing on. Practically, this can get messy, especially if you're on Windows, where file locking is also likely to cause issues.

So sure, if you want to be picky, "Grandfather paradox" is probably the wrong analogy. Pull requests are welcome.

3.11 Contact

If you have questions, concerns, ideas about the product or how to make it better, please let us know!

Here are some ways to get in contact with us and other KSCONF users:

- Chat about `#ksconf` on Splunk's Slack channel.
- Discuss features or ask general questions in GitHub discussions. **This is new**, please drop by and let us know if this is helpful or not.
- Email us at hello@kintyre.co for general inquiries, if you're interested in commercial support, or would like to fund new features.
- Ask a question on
  - GitHub

3.12 Command line reference

KSCONF supports the following CLI options:

3.12.1 ksconf

```
usage: ksconf [-h] [--version] [--force-color] [--disable-color]
              {attr-get,attr-set,check,combine,diff,filter,merge,minimize,
↪package,promote,rest-export,rest-publish,snapshot,sort,unarchive,xml-
↪format}
              ...

KSCONF: Ksconf Splunk CONFig tool

This utility handles a number of common Splunk app maintenance tasks in a␣
↪small
and easy to deploy package.  Specifically, this tool deals with many of the
```

(continues on next page)

(continued from previous page)

```
nuances with storing Splunk apps in git and pointing live Splunk apps to a␣
↪git
repository.  Merging changes from the live system's (local) folder to the
version controlled (default) folder and dealing with more than one layer of
"default" are all supported tasks which are not native to Splunk.

positional arguments:
  {attr-get,attr-set,check,combine,diff,filter,merge,minimize,package,
↪promote,rest-export,rest-publish,snapshot,sort,unarchive,xml-format}
    attr-get            Get the value from a specific stanzas and attribute
    attr-set            Set the value of a specific stanzas and attribute
    check               Perform basic syntax and sanity checks on .conf␣
↪files
    combine             Combine configuration files across multiple source
                        directories into a single destination directory.␣
↪This
                        allows for an arbitrary number of Splunk␣
↪configuration
                        layers to coexist within a single app. Useful in␣
↪both
                        ongoing merge and one-time ad-hoc use.
    diff                Compare settings differences between two .conf␣
↪files
                        ignoring spacing and sort order
    filter              A stanza-aware GREP tool for conf files
    merge               Merge two or more .conf files
    minimize            Minimize the target file by removing entries
                        duplicated in the default conf(s)
    package             Create a Splunk app .spl file from a source␣
↪directory
    promote             Promote .conf settings between layers using either
                        batch or interactive mode. Frequently this is used␣
↪to
                        promote conf changes made via the UI (stored in the
                        'local' folder) to a version-controlled directory,
                        such as 'default'.
    rest-export         Export .conf settings as a curl script to apply to␣
↪a
                        Splunk instance later (via REST)
    rest-publish        Publish .conf settings to a live Splunk instance␣
↪via
                        REST
    snapshot            Snapshot .conf file directories into a JSON dump
                        format
    sort                Sort a Splunk .conf file creating a normalized␣
↪format
```

(continues on next page)

(continued from previous page)

```
                        appropriate for version control
    unarchive           Install or upgrade an existing app in a git-
↪friendly
                        and safe way
    xml-format          Normalize XML view and nav files

options:
  -h, --help            show this help message and exit
  --version             show program's version number and exit
  --force-color         Force TTY color mode on. Useful if piping the␣
↪output a
                        color-aware pager, like 'less -R'
  --disable-color       Disable TTY color mode. This can also be setup as
                        environmental variable: 'export KSCONF_TTY_
↪COLOR=off'
```

3.12.2 ksconf attr-get

```
usage: ksconf attr-get [-h] --stanza STANZA --attribute ATTR [--missing-
↪okay]
                       [-o OUTPUT]
                       conf [conf ...]

Get a specific stanza and attribute value from a Splunk .conf file.

positional arguments:
  conf                  Input file or standard input.

options:
  -h, --help            show this help message and exit
  --stanza STANZA, -s STANZA
                        Name of the stanza within CONF to retrieve.
  --attribute ATTR, --attr ATTR, -a ATTR
                        Name of attribute within STANZA to retrieve.
  --missing-okay        Ignore missing stanzas and attributes.
  -o OUTPUT, --output OUTPUT
                        File where the filtered results are written.␣
↪Defaults
                        to standard out.
```

3.12.3 ksconf attr-set

```
usage: ksconf attr-set [-h] --stanza STANZA --attribute ATTR
                       [--value-type TYPE] [--create-missing] [--no-
↪overwrite]
                       conf value

Set a specific stanza and attribute value of a Splunk .conf file.
The value can be provided as a command line argument, file, or
environment variable

This command does not support preserving leading or trailing whitespace.
Normally this is desireable.

positional arguments:
  conf                  Configuration file to update.
  value                 Value to apply to the conf file. Note that this␣
↪can be
                        a raw text string, or the name of the file, or an
                        environment variable

options:
  -h, --help            show this help message and exit
  --stanza STANZA, -s STANZA
                        Name of the stanza within CONF to set.
  --attribute ATTR, --attr ATTR, -a ATTR
                        Name of the attribute within STANZA to set.
  --value-type TYPE, -t TYPE
                        Select the type of VALUE. The default is a string.
                        Alternatively, the real value can be provided␣
↪within a
                        file, or an environment variable.
  --create-missing      Create a new conf file if it doesn't currently␣
↪exist.
  --no-overwrite        Only set VALUE if none currently exists. This can␣
↪be
                        used to safely set a one-time default, but don't
                        update overwrite an existing value.
```

3.12.4 ksconf check

```
usage: ksconf check [-h] [--quiet] FILE [FILE ...]

Provides basic syntax and sanity checking for Splunk's .conf files. Use
Splunk's built-in 'btool check' for a more robust validation of attributes␣
↪and
values. Consider using this utility as part of a pre-commit hook.

positional arguments:
  FILE         One or more configuration files to check. If '-' is given,␣
↪then
               read a list of files to validate from standard input

options:
  -h, --help   show this help message and exit
  --quiet, -q  Reduce the volume of output.
```

3.12.5 ksconf combine

```
usage: ksconf combine [-h] --target TARGET [-m {auto,dir.d,disable}] [-q]
                      [-I PATTERN] [-E PATTERN] [--enable-handler {jinja}]
                      [--template-vars TEMPLATE_VARS] [--dry-run]
                      [--follow-symlink] [--banner BANNER] [-K KEEP_
↪EXISTING]
                      [--disable-marker] [--disable-cleanup]
                      source [source ...]

Merge .conf settings from multiple source directories into a combined␣
↪target
directory.  Configuration files can be stored in a '/etc/*.d' like␣
↪directory
structure and consolidated back into a single 'default' directory.

This command supports both one-time operations and recurring merge jobs. ␣
↪For
example, this command can be used to combine all users' knowledge objects␣
↪(stored
in 'etc/users') after a server migration, or to merge a single user's␣
↪settings
after their account has been renamed.  Recurring operations assume some␣
↪type
of external scheduler is being used.  A best-effort is made to only write␣
↪to
target files as needed.
```

(continues on next page)

(continued from previous page)

```
The 'combine' command takes your logical layers of configs (upstream,␣
↪corporate,
Splunk admin fixes, and power user knowledge objects, ...) expressed as
individual folders and merges them all back into the single 'default'␣
↪folder
that Splunk reads from.  One way to keep the 'default' folder up-to-date is
using client-side git hooks.

No directory layout is mandatory, but taking advantages of the native-
↪support
for 'dir.d' layout works well for many uses cases.  This idea is borrowed␣
↪from
the Unix System V concept where many services natively read their config␣
↪files
from '/etc/*.d' directories.

Version notes:  dir.d was added in ksconf 0.8.  Starting in 1.0 the␣
↪default will
switch to 'dir.d', so if you need the old behavior be sure to update your␣
↪scripts.

positional arguments:
  source                The source directory where configuration files␣
↪will be
                        merged from. When multiple source directories are
                        provided, start with the most general and end with␣
↪the
                        specific; later sources will override values from␣
↪the
                        earlier ones. Supports wildcards so a typical Unix
                        'conf.d/##-NAME' directory structure works well.

options:
  -h, --help            show this help message and exit
  --target TARGET, -t TARGET
                        Directory where the merged files will be stored.
                        Typically either 'default' or 'local'
  -m {auto,dir.d,disable}, --layer-method {auto,dir.d,disable}
                        Set the layer type used by SOURCE. Use 'dir.d' if␣
↪you
                        have directories like 'MyApp/default.d/##-layer-
↪name',
                        or use 'disable' to manage layers explicitly and␣
↪avoid
                        any accidental layer detection. By default, 'auto'
```

(continues on next page)

(continued from previous page)

```
                        mode will enable transparent switching between
↪'dir.d'
                        and 'disable' (legacy) behavior, however this␣
↪option
                        will be removed in a future release.
  -q, --quiet           Make output a bit less noisy. This may change in␣
↪the
                        future...
  -I PATTERN, --include PATTERN
                        Name or pattern of layers to include.
  -E PATTERN, --exclude PATTERN
                        Name or pattern of layers to exclude from the␣
↪target.
  --enable-handler {jinja}
                        Enable optional file handling support
  --template-vars TEMPLATE_VARS
                        Set template variables as key=value or YAML/JSON,␣
↪if
                        filename prepend with @
  --dry-run, -D         Enable dry-run mode. Instead of writing to TARGET,
                        preview changes as a 'diff'. If TARGET doesn't␣
↪exist,
                        then show the merged file.
  --follow-symlink, -l  Follow symbolic links pointing to directories.
                        Symlinks to files are always followed.
  --banner BANNER, -b BANNER
                        A banner or warning comment added to the top of the
                        TARGET file. Used to discourage Splunk admins from
                        editing an auto-generated file.
  -K KEEP_EXISTING, --keep-existing KEEP_EXISTING
                        Existing file(s) to preserve in the TARGET folder.
                        This argument may be used multiple times.
  --disable-marker      Prevents the creation of or checking for the
                        '.ksconf_controlled' marker file safety check. This
                        file is typically used indicate that the␣
↪destination
                        folder is managed by ksconf. This option should be
                        reserved for well-controlled batch processing
                        scenarios.
  --disable-cleanup     Disable all file removal operations. Skip the␣
↪cleanup
                        phase that typically removes files in TARGET that␣
↪no
                        longer exist in SOURCE
```

3.12.6 ksconf diff

```
usage: ksconf diff [-h] [-o FILE] [--detail {global,stanza,key}] [--
↪comments]
                   [--format {diff,json}]
                   CONF1 CONF2

Compares the content differences of two .conf files

This command ignores textual differences (like order, spacing, and␣
↪comments) and
focuses strictly on comparing stanzas, keys, and values.  Note that spaces␣
↪within
any given value, will be compared. Multi-line fields are compared in a␣
↪more traditional
'diff' output so that long saved searches and macros can be compared more␣
↪easily.

positional arguments:
  CONF1                 Left side of the comparison
  CONF2                 Right side of the comparison

options:
  -h, --help            show this help message and exit
  -o FILE, --output FILE
                        File where difference is stored. Defaults to␣
↪standard
                        out.
  --detail {global,stanza,key}, -d {global,stanza,key}
                        Control the highest level for which 'replace'␣
↪events
                        may occur.
  --comments, -C        Enable comparison of comments. (Unlikely to work
                        consistently)
  --format {diff,json}, -f {diff,json}
                        Output file format to produce. 'diff' the the␣
↪classic
                        format used by default. 'json' is helpful when␣
↪trying
                        to review changes programmatically.
```

3.12.7 ksconf filter

```
usage: ksconf filter [-h] [-o FILE] [--comments] [--verbose] [--skip-
↪broken]
                     [--match {regex,wildcard,string}] [--ignore-case]
                     [--invert-match] [--files-with-matches]
                     [--count | --brief] [--stanza PATTERN]
                     [--attr-present ATTR] [--attr-matches ATTR PATTERN]
                     [--attr-not-matches ATTR PATTERN] [-e | -d]
                     [--keep-attrs WC-ATTR] [--reject-attrs WC-ATTR]
                     CONF [CONF ...]

Filter the contents of a conf file in various ways. Stanzas can be␣
↪included or
excluded based on a provided filter or based on the presence or value of a
key. Where possible, this command supports GREP-like arguments to bring a
familiar feel.

positional arguments:
  CONF                  Input conf file

options:
  -h, --help            show this help message and exit
  -o FILE, --output FILE
                        File where the filtered results are written.␣
↪Defaults
                        to standard out.
  --comments, -C        Preserve comments. Comments are discarded by␣
↪default.
  --verbose             Enable additional output.
  --skip-broken         Skip broken input files. Without this things like
                        duplicate stanzas and invalid entries will cause
                        processing to stop.
  --match {regex,wildcard,string}, -m {regex,wildcard,string}
                        Specify pattern matching mode. Defaults to
↪'wildcard'
                        allowing for '*' and '?' matching. Use 'regex' for
                        more power but watch out for shell escaping. Use
                        'string' to enable literal matching.
  --ignore-case, -i     Ignore case when comparing or matching strings. By
                        default matches are case-sensitive.
  --invert-match, -v    Invert match results. This can be used to show what
                        content does NOT match, or make a backup copy of
                        excluded content.

Output mode:
```

(continues on next page)

(continued from previous page)

```
  Select an alternate output mode. If any of the following options are␣
↪used,
  the stanza output is not shown.

  --files-with-matches, -l
                        List files that match the given search criteria
  --count, -c           Count matching stanzas
  --brief, -b           List name of matching stanzas

Stanza selection:
  Include or exclude entire stanzas using these filter options. All filter
  options can be provided multiple times. If you have a long list of
  filters, they can be saved in a file and referenced using the special
  'file://' prefix. One entry per line. Entries can be either a literal
  strings, wildcards, or regexes, depending on MATCH.

  --stanza PATTERN      Match any stanza who's name matches the given␣
↪pattern.
                        PATTERN supports bulk patterns via the 'file://'
                        prefix.
  --attr-present ATTR   Match any stanza that includes the ATTR attribute.
                        ATTR supports bulk attribute patterns via the
                        'file://' prefix.
  --attr-matches ATTR PATTERN, --attr-eq ATTR PATTERN
                        Match any stanza containing ATTR == PATTERN.␣
↪PATTERN
                        supports the special 'file://filename' syntax.
                        Matching can be a direct string comparison␣
↪(equals),
                        or a regex and wildcard match. Note that all '--
↪attr-
                        match' and '--attr-not-match' arguments are matched
                        together. For a stanza to match, all rules must␣
↪apply.
                        If attr is missing from a stanza, the value␣
↪becomes an
                        empty string for matching purposes.
  --attr-not-matches ATTR PATTERN, --attr-ne ATTR PATTERN
                        Match any stanza containing ATTR != PATTERN. See '-
↪-
                        attr-matches' for additional details.
  -e, --enabled-only    Keep only enabled stanzas. Any stanza containing
                        'disabled = 1' will be removed. The value of
                        'disabled' is assumed to be false by default.
  -d, --disabled-only   Keep disabled stanzas only. The value of the
                        `disabled` attribute is interpreted as a boolean.
```

(continues on next page)

(continued from previous page)

```
Attribute selection:
  Include or exclude attributes passed through. By default, all attributes
  are preserved. Allowlist (keep) operations are preformed before blocklist
  (reject) operations.

  --keep-attrs WC-ATTR  Select which attribute(s) will be preserved. This
                        space separated list of attributes indicates what␣
↪to
                        preserve. Supports wildcards.
  --reject-attrs WC-ATTR
                        Select which attribute(s) will be discarded. This
                        space separated list of attributes indicates what␣
↪to
                        discard. Supports wildcards.
```

3.12.8 ksconf merge

```
usage: ksconf merge [-h] [--target TARGET] [--ignore-missing] [--in-place]
                    [--dry-run] [--banner BANNER]
                    conf [conf ...]

Merge two or more .conf files into a single combined .conf file. This is
similar to the way that Splunk logically combines the 'default' and 'local'
folders at runtime.

positional arguments:
  conf                  The source configuration file(s) to collect␣
↪settings
                        from.

options:
  -h, --help            show this help message and exit
  --target TARGET, -t TARGET
                        Destination file for merged configurations. If not
                        provided, the merged conf is written to standard
                        output.
  --ignore-missing, -s  Silently ignore any missing CONF files.
  --in-place, -i        Enable in-place update mode. When selected, the␣
↪TARGET
                        file will also be considered as the base of the␣
↪merge
                        operation. All CONF files will be merged with␣
↪TARGET.
```

(continues on next page)

(continued from previous page)

```
                        When disabled, any existing content within TARGET␣
↪is
                        ignored and overwritten.
  --dry-run, -D         Enable dry-run mode. Instead of writing to TARGET,
                        preview changes in 'diff' format. If TARGET doesn't
                        exist, then show the merged file.
  --banner BANNER, -b BANNER
                        A banner or warning comment added to the top of the
                        TARGET file. Used to discourage Splunk admins from
                        editing an auto-generated file.
```

3.12.9 ksconf minimize

```
usage: ksconf minimize [-h] [--target TARGET] [--dry-run | --output OUTPUT]
                       [--explode-default] [-k PRESERVE_KEY]
                       CONF [CONF ...]

Minimize a conf file by removing any duplicated default settings. Reduce a
local conf file to only your intended changes without manually tracking␣
↪which
entries you've edited. Minimizing local conf files makes your local
customizations easier to read and often results in cleaner upgrades.

positional arguments:
  CONF                  The default configuration file(s) used to determine
                        what base settings are. The base settings determine
                        what is unnecessary to repeat in target file.

options:
  -h, --help            show this help message and exit
  --target TARGET, -t TARGET
                        The local file that you wish to remove duplicate
                        settings from. This file will be read from and then
                        replaced with a minimized version.
  --dry-run, -D         Enable dry-run mode. Instead of writing and␣
↪minimizing
                        the TARGET file, preview what would be removed as a
                        'diff'.
  --output OUTPUT       Write the minimized output to a separate file␣
↪instead
                        of updating TARGET.
  --explode-default, -E
                        Enable minimization across stanzas for special use-
                        cases. Helpful when dealing with stanzas downloaded
```

(continues on next page)

(continued from previous page)

```
                        from a REST endpoint or 'btool list' output.
  -k PRESERVE_KEY, --preserve-key PRESERVE_KEY
                        Specify attributes that should always be kept.
```

3.12.10 ksconf package

```
usage: ksconf package [-h] [-f SPL] [--app-name APP_NAME]
                      [--blocklist BLOCKLIST] [--allowlist ALLOWLIST]
                      [--layer-method {dir.d,disable}] [-I PATTERN]
                      [-E PATTERN] [--enable-handler {jinja}]
                      [--template-vars TEMPLATE_VARS] [--follow-symlink]
                      [--set-version VERSION] [--set-build BUILD]
                      [--allow-local | --block-local | --merge-local]
                      [--release-file RELEASE_FILE]
                      SOURCE

Create a Splunk app or add on tarball ('.spl') file from an app directory.
'ksconf package' can do useful things like, exclude unwanted files, combine
layers, set the application version and build number, drop or promote the
'local' directory into 'default'. Note that some arguments, like the 'FILE'
support special values that can be automatically evaluated at runtime. For
example the placeholders '{{version}}' or '{{git_tag}}' can be expanded␣
↪into
the output tarball filename. If both layering and templating are in use at␣
↪the
same time, be aware that templates are rendered prior to layering␣
↪operations.
This allows, for example, one layer to include a simple 'indexes.conf' file
and another layer to include an 'indexes.conf.j2' template.

positional arguments:
  SOURCE                Source directory for the Splunk app.

options:
  -h, --help            show this help message and exit
  -f SPL, --file SPL    Name of splunk app file (tarball) to create.
                        Placeholder variables in '{{var}}' syntax can be␣
↪used
                        here.
  --app-name APP_NAME   Specify the top-level app folder name. If this is␣
↪not
                        given, the app folder name is automatically␣
↪extracted
                        from the basename of SOURCE. Placeholder variables,
```

(continues on next page)

(continued from previous page)

```
                        such as '{{app_id}}' can be used here.
  --blocklist BLOCKLIST, -b BLOCKLIST
                        Pattern for files/directories to exclude. Can be␣
↪given
                        multiple times. You can load multiple exclusions␣
↪from
                        disk by using 'file://path' which can be used with
                        '.gitignore' for example. (Default includes: '.git*
↪',
                        '*.py[co]', '__pycache__', '.DS_Store')
  --allowlist ALLOWLIST, -a ALLOWLIST
                        Remove a pattern that was previously added to the
                        blocklist.
  --enable-handler {jinja}
                        Enable optional file handling support
  --template-vars TEMPLATE_VARS
                        Set template variables as key=value or YAML/JSON,␣
↪if
                        filename prepend with @
  --follow-symlink, -l  Follow symbolic links pointing to directories.
                        Symlinks to files are always followed.
  --set-version VERSION
                        Set application version. By default the application
                        version is read from default/app.conf. Placeholder
                        variables such as '{{git_tag}}' can be used here.
  --set-build BUILD     Set application build number.
  --allow-local         Allow the 'local' folder to be kept as-is WARNING:
                        This goes against Splunk packaging practices, and␣
↪will
                        cause AppInspect to fail. However, this option can␣
↪be
                        useful for private package transfers between␣
↪servers,
                        app backups, or other admin-like tasks.
  --block-local         Block the 'local' folder and 'local.meta' from the
                        package.
  --merge-local         Merge any files in 'local' into the 'default'␣
↪folder
                        during packaging. This is the default behavior.

Layer filtering:
  If the app being packaged includes multiple layers, these arguments can␣
↪be
  used to control which ones should be included in the final app file. If␣
↪no
  layer options are specified, then all layers will be included.
```

(continues on next page)

(continued from previous page)

```
  --layer-method {dir.d,disable}
                        Set the layer type used by SOURCE. Additional
                        description provided in in the 'combine' command.
  -I PATTERN, --include PATTERN
                        Name or pattern of layers to include.
  -E PATTERN, --exclude PATTERN
                        Name or pattern of layers to exclude from the␣
↪target.

Advanced Build Options:
  The following options are for more advanced app building workflows.

  --release-file RELEASE_FILE
                        Write the path of the newly generated archive file
                        (SPL) after the archive is written. This is useful␣
↪in
                        build scripts when the SPL contains variables so␣
↪the
                        final name may not be known ahead of time.
```

3.12.11 ksconf promote

```
usage: ksconf promote [-h] [--batch | --interactive | --summary | --diff]
                      [--verbose] [--match {regex,wildcard,string}]
                      [--ignore-case] [--invert-match] [--stanza PATTERN]
                      [--force] [--keep] [--keep-empty]
                      SOURCE TARGET

Propagate .conf settings applied in one file to another.  Typically this␣
↪is used
to move 'local' changes (made via the UI) into another layer, such as the
'default' or a named 'default.d/50-xxxxx') folder.

Promote has two modes:  batch and interactive.  In batch mode, all changes␣
↪are
applied automatically and the (now empty) source file is removed.  In␣
↪interactive
mode, the user is prompted to select stanzas to promote.  This way local␣
↪changes
can be held without being promoted.

NOTE: Changes are *MOVED* not copied, unless '--keep' is used.
```

(continues on next page)

(continued from previous page)

```
positional arguments:
  SOURCE                The source configuration file to pull changes from.
                        (Typically the 'local' conf file)
  TARGET                Configuration file or directory to push the changes
                        into. (Typically the 'default' folder)

options:
  -h, --help            show this help message and exit
  --batch, -b           Use batch mode where all configuration settings are
                        automatically promoted. All changes are removed␣
↪from
                        source and applied to target. The source file will␣
↪be
                        removed unless '--keep-empty' is used.
  --interactive, -i     Enable interactive mode where the user will be
                        prompted to approve the promotion of specific␣
↪stanzas
                        and attributes. The user will be able to apply,␣
↪skip,
                        or edit the changes being promoted.
  --summary, -s         Summarize content that could be promoted.
  --diff, -d            Show the diff of what would be promoted.
  --verbose             Enable additional output.
  --force, -f           Disable safety checks. Don't check to see if SOURCE
                        and TARGET share the same basename.
  --keep, -k            Keep conf settings in the source file. All changes
                        will be copied into the TARGET file instead of␣
↪being
                        moved there. This is typically a bad idea since␣
↪local
                        always overrides default.
  --keep-empty          Keep the source file, even if after the settings
                        promotions the file has no content. By default,␣
↪SOURCE
                        will be removed after all content has been moved␣
↪into
                        TARGET. Splunk will re-create any necessary local
                        files on the fly.

Automatic filtering options:
  Include or exclude stanzas to promote using these filter options.
  Stanzas selected by these filters will be promoted.

  All filter options can be provided multiple times.
  If you have a long list of filters, they can be saved in a file and
  referenced using the special 'file://' prefix.  One entry per line.
```

(continues on next page)

(continued from previous page)

```
  --match {regex,wildcard,string}, -m {regex,wildcard,string}
                        Specify pattern matching mode. Defaults to
↪'wildcard'
                        allowing for '*' and '?' matching. Use 'regex' for
                        more power but watch out for shell escaping. Use
                        'string' to enable literal matching.
  --ignore-case         Ignore case when comparing or matching strings. By
                        default matches are case-sensitive.
  --invert-match, -v    Invert match results. This can be used to prevent
                        content from being promoted.
  --stanza PATTERN      Promote any stanza with a name matching the given
                        pattern. PATTERN supports bulk patterns via the
                        'file://' prefix.
```

3.12.12 ksconf rest-export

```
usage: ksconf rest-export [-h] [--output FILE] [--disable-auth-output]
                          [--pretty-print] [-u | -D] [--url URL] [--app␣
↪APP]
                          [--user USER] [--owner OWNER] [--conf TYPE]
                          [--extra-args EXTRA_ARGS]
                          CONF [CONF ...]

Build an executable script of the stanzas in a configuration file that can␣
↪be later applied to
a running Splunk instance via the Splunkd REST endpoint.

This can be helpful when pushing complex props and transforms to an␣
↪instance where you only have
UI access and can't directly publish an app.

positional arguments:
  CONF                  Configuration file(s) to export settings from.

options:
  -h, --help            show this help message and exit
  --output FILE, -t FILE
                        Save the shell script output to this file. If not
                        provided, the output is written to standard output.
  -u, --update          Assume that the REST entities already exist. By
                        default, output assumes stanzas are being created.
  -D, --delete          Remove existing REST entities. This is a␣
↪destructive
```

(continues on next page)

(continued from previous page)

```
                        operation. In this mode, stanza attributes are
                        unnecessary and ignored. NOTE: This works for
↪'local'
                        entities only; the default folder cannot be␣
↪updated.
  --url URL             URL of Splunkd. Default: https://localhost:8089
  --app APP             Set the namespace (app name) for the endpoint
  --user USER           Deprecated. Use --owner instead.
  --owner OWNER         Set the object owner. Typically, the default of
                        'nobody' is ideal if you want to share the
                        configurations at the app-level.
  --conf TYPE           Explicitly set the configuration file type. By
                        default, this is derived from CONF, but sometimes␣
↪it's
                        helpful to set this explicitly. Can be any valid
                        Splunk conf file type. Examples include: 'app',
                        'props', 'tags', 'savedsearches', etc.
  --extra-args EXTRA_ARGS
                        Extra arguments to pass to all CURL commands. Quote
                        arguments on the command line to prevent confusion
                        between arguments to ksconf vs curl.

Output Control:
  --disable-auth-output
                        Turn off sample login curl commands from the␣
↪output.
  --pretty-print, -p    Enable pretty-printing. Make shell output a bit␣
↪more
                        readable by splitting entries across lines.
```

3.12.13 ksconf rest-publish

```
usage: ksconf rest-publish [-h] [--conf TYPE] [-m META] [--url URL]
                           [--user USER] [--pass PASSWORD] [-k]
                           [--session-key SESSION_KEY] [--app APP]
                           [--owner OWNER] [--sharing {user,app,global}] [-
↪D]
                           CONF [CONF ...]

Publish stanzas in a .conf file to a running Splunk instance via REST. This
requires access to the HTTPS endpoint of Splunk. By default, ksconf will
handle both the creation of new stanzas and the update of existing stanzas.
This can be used to push full configuration stanzas where you only have␣
↪REST
```

(continues on next page)

(continued from previous page)

```
access and can't directly publish an app. Only attributes present in the␣
↪conf
file are pushed. While this may seem obvious, this fact can have profound
implications in certain situations, like when using this command for
continuous updates. This means that it's possible for the source .conf to
ultimately differ from what ends up on the server's .conf file. One way to
avoid this, is to explicitly remove an object using '--delete' mode first,␣
↪and
then insert a new copy of the object. Of course, this means that the object
will be unavailable. The other impact is that diffs only compares and␣
↪shows a
subset of attribute. Be aware, that for consistency, the configs/conf-TYPE
endpoint is used for this command. Therefore, a reload may be required for␣
↪the
server to use the published config settings.

positional arguments:
  CONF                  Configuration file(s) to export settings from.

options:
  -h, --help            show this help message and exit
  --conf TYPE           Explicitly set the configuration file type. By
                        default, this is derived from CONF, but sometimes␣
↪it's
                        helpful to set this explicitly. Can be any valid
                        Splunk conf file type. Examples include: 'app',
                        'props', 'tags', 'savedsearches', etc.
  -m META, --meta META  Specify one or more '.meta' files to determine the
                        desired read & write ACLs, owner, and sharing for
                        objects in the CONF file.
  --url URL             URL of Splunkd. Default: https://localhost:8089
  --user USER           Login username Splunkd. Default: admin
  --pass PASSWORD       Login password Splunkd. Default: changeme
  -k, --insecure        Disable SSL cert validation.
  --session-key SESSION_KEY
                        Use an existing session token instead of using a
                        username and password to login.
  --app APP             Set the namespace (app name) for the endpoint
  --owner OWNER         Set the user who owns the content. The default of
                        'nobody' works well for app-level sharing.
  --sharing {user,app,global}
                        Set the sharing mode.
  -D, --delete          Remove existing REST entities. This is a␣
↪destructive
                        operation. In this mode, stanza attributes are
                        unnecessary. NOTE: This works for 'local' entities
```

(continues on next page)

(continued from previous page)

```
                        only; the default folder cannot be updated.
```

3.12.14 ksconf snapshot

```
usage: ksconf snapshot [-h] [--output FILE] [--minimize] PATH [PATH ...]

Build a static snapshot of various configuration files stored within a
structured json export format. If the .conf files being captured are␣
↪within a
standard Splunk directory structure, then certain metadata and namespace
information is assumed based on typical path locations. Individual apps or
conf files can be collected as well, but less metadata may be extracted.

positional arguments:
  PATH                  Directory from which to load configuration files.␣
↪All
                        .conf and .meta file are included recursively.

options:
  -h, --help            show this help message and exit
  --output FILE, -o FILE
                        Save the snapshot to the named files. If not␣
↪provided,
                        the snapshot is written to standard output.
  --minimize            Reduce the size of the JSON output by removing
                        whitespace. Reduces readability.
```

3.12.15 ksconf sort

```
usage: ksconf sort [-h] [--target FILE | --inplace] [-F] [-q] [-n LINES]
                   FILE [FILE ...]

Sort a Splunk .conf file.  Sort has two modes:  (1) by default, the sorted
config file will be echoed to the screen.  (2) the config files are updated
in-place when the '-i' option is used.

Manually managed conf files can be protected against changes by adding a␣
↪comment containing the
string 'KSCONF-NO-SORT' to the top of any .conf file.

positional arguments:
  FILE                  Input file to sort, or standard input.
```

(continues on next page)

(continued from previous page)

```
options:
  -h, --help            show this help message and exit
  --target FILE, -t FILE
                        File to write results to. Defaults to standard␣
↪output.
  --inplace, -i         Replace the input file with a sorted version.␣
↪WARNING:
                        This a potentially destructive operation that may
                        move/remove comments.
  -n LINES, --newlines LINES
                        Number of lines between stanzas.

In-place update arguments:
  -F, --force           Force file sorting for all files, even for files
                        containing the special 'KSCONF-NO-SORT' marker.
  -q, --quiet           Reduce the output. Reports only updated or invalid
                        files. This is useful for pre-commit hooks, for
                        example.
```

3.12.16 ksconf unarchive

```
usage: ksconf unarchive [-h] [--dest DIR] [--app-name NAME]
                        [--default-dir DIR] [--exclude EXCLUDE] [--keep␣
↪KEEP]
                        [--allow-local]
                        [--git-sanity-check {off,changed,untracked,ignored}
↪]
                        [--git-mode {nochange,stage,commit}] [--no-edit]
                        [--git-commit-args GIT_COMMIT_ARGS]
                        SPL

Install or overwrite an existing app in a git-friendly way.
If the app already exists, steps will be taken to upgrade it safely.

The 'default' folder can be redirected to another path (i.e., 'default.d/
↪10-upstream' or
other desirable path if you're using the 'ksconf combine' tool to manage␣
↪extra layers).

positional arguments:
  SPL                   The path to the archive to install.

options:
  -h, --help            show this help message and exit
```

(continues on next page)

(continued from previous page)

```
  --dest DIR            Set the destination path where the archive will be
                        extracted. By default, the current directory is␣
↪used.
                        Sane values include: etc/apps, etc/deployment-apps,
                        and so on.
  --app-name NAME       The app name to use when expanding the archive. By
                        default, the app name is taken from the archive as␣
↪the
                        top-level path included in the archive (by
                        convention).
  --default-dir DIR     Name of the directory where the default contents␣
↪will
                        be stored. This is a useful feature for apps that␣
↪use
                        a dynamic default directory that's created and␣
↪managed
                        by the 'combine' mode.
  --exclude EXCLUDE, -e EXCLUDE
                        Add a file pattern to exclude from extraction.
                        Splunk's pseudo-glob patterns are supported here.
↪'*'
                        for any non-directory match, '...' for ANY␣
↪(including
                        directories), and '?' for a single character.
  --keep KEEP, -k KEEP  Specify a pattern for files to preserve during an
                        upgrade. Repeat this argument to keep multiple
                        patterns.
  --allow-local         Allow local/* and local.meta files to be extracted
                        from the archive.
  --git-sanity-check {off,changed,untracked,ignored}
                        By default, 'git status' is run on the destination
                        folder to detect working tree or index␣
↪modifications
                        before the unarchive process start. Sanity check
                        choices go from least restrictive to most thorough:
                        'off' prevents all safety checks. 'changed' aborts
                        only upon local modifications to files tracked by␣
↪git.
                        'untracked' (the default) looks for changed and
                        untracked files. 'ignored' aborts is (any) local
                        changes, untracked, or ignored files are found.
  --git-mode {nochange,stage,commit}
                        Set the desired level of git integration. The␣
↪default
                        mode is *stage*, where new, updated, or removed␣
↪files
```

(continues on next page)

(continued from previous page)

```
                        are automatically handled for you. To prevent any
↪'git
                        add' or 'git rm' commands from being run, pick the
                        'nochange' mode.
  --no-edit             Tell git to skip opening your editor on commit. By
                        default, you will be prompted to review/edit the
                        commit message. (Git Tip: Delete the content of the
                        default message to abort the commit.)
  --git-commit-args GIT_COMMIT_ARGS, -G GIT_COMMIT_ARGS
                        Extra arguments to pass to 'git'
```

3.12.17 ksconf xml-format

```
usage: ksconf xml-format [-h] [--indent INDENT] [--quiet] FILE [FILE ...]

Normalize and apply consistent XML indentation and CDATA usage for XML
dashboards and navigation files. Technically this could be used on *any*␣
↪XML
file, but certain element names specific to Splunk's simple XML dashboards␣
↪are
handled specially, and therefore could result in unusable results. The
expected indentation level is guessed based on the first element␣
↪indentation,
but can be explicitly set if not detectable.

positional arguments:
  FILE              One or more XML files to check. If '-' is given, then a
                    list of files is read from standard input

options:
  -h, --help        show this help message and exit
  --indent INDENT   Number of spaces. This is only used if indentation␣
↪cannot
                    be guessed from the existing file.
  --quiet, -q       Reduce the volume of output.
```

3.13 Changelog

Note: Changes in the *devel* branch, but not released yet are marked as *DRAFT*.

3.13.1 Ksconf 0.13

Switching to use Python's namespace packages. This is an internal change that should make future expansions easier but should have no impact on normal users.

New namespaces:

- `ksconf`
- `ksconf.commands`
- `ksconf.plugins`

Renames:

- `ksconf/__init__` -> `ksconf._ksconf`
- `ksconf/commands/__init__.py` -> `ksconf/command.py`

Ksconf v0.13.1 (2023-10-05)

- Removed overlooked debug message at startup.

Ksconf v0.13.0 (2023-10-05)

- Switching to python package namespaces for for `ksconf` and `ksconf.commands` and created `ksconf.plugins`. This allows for more flexible packaging of various ksconf components including optional subcommands and plugins. Unless you are a python developer, you should never notice a difference. Splunk App users should re-install to avoid any confusion.

3.13.2 Ksconf 0.12

Highlights:

- Add new ksconf subcommands for very basic, but previously missing, *ksconf attr-get* and *ksconf attr-set*. These commands allow for easy target information for capture or update.
- Pre-commit hooks have been moved into their own *ksconf-pre-commit repo*. To allow time for migration to the new repo, the existing hooks will remain for a few release before being removed. To migrate, simply add `-pre-commit` to the end of the `repo` field, and update `rev` to `v0.12.0` or later.

Packaging changes:

- Dropped hard `lxml` from requirements. This is still handled automatically when using the pre-commit hooks (from the new repository). But this may be missing. To get access to all CLI functionality, run `pip install ksconf[thirdparty]`, or for the full experience use `pip install ksconf[fully-loaded]`
- Remove the use of the `endpoints` python package and shift to using `importlib.metadata` (or the equivalent backport), as it suggested by the original author of that package. The original necessity of this library was to workaround performance issues in `pkg_resources` (and the fact that it's no present in Splunk's embedded python. This move reduces code complexity but it does mean some additional runtime dependencies on older versions of Python. In many cases, this really isn't a new dependency, since pluggy requires it as well.

Ksconf v0.12.2 (2023-10-05)

- *App building with Ksconf*: Added a convenience method to allow running ksconf commands more easily during the build process. You can now invoke ksconf using `run_ksconf()` method which allows direct execution of a ksconf command. Previously this was accomplished by using `run()`, using Python interpreter internal path as the executable, launching the ksconf in "module" mode. So this approach is simpler and in the future it may be invoked internally, removing the need for launching an additional Python process.
- Enhanced plugin error handling.
- Many little doc build fixes.

Ksconf v0.12.1 (2023-10-03)

- Introducing *ksconf attr-get* and *ksconf attr-set* - the newest and simplest ksconf commands ever! Use this to quickly grab and/or update a specific stanzas, attribute combination from a conf file.
- Add new attribute-level matching logic to `ksconf filter`. Use `--attr-matches` and/or `--attr-not-matches` to match specific attribute and value combinations for stanza matching. This can be used to find props with a specific `KV_MODE`, find saved search containing a specific search command, or list indexes not using `volume:` designation. See the *ksconf filter* docs for example usage. Thanks to yohonet for inspiring this change, along with the new `attr-get` command.
- Add `--in-place` processing behavior for *ksconf merge* to simplify the process of merging new content into an existing conf file.
- Docs Improvements: * Fixed documentation generation bug that prevented command line options from showing up in the per-command doc pages. (Broken since v0.10) * Fixed docs embedded in the Splunk app (and possibly causing some other display issues on the main rtd site)
- Fixed some CLI file handling bug that resulted in broken use of - (stdin) and/or fancy shell commands involving `<(some command)` syntax, which can be a helpful trick to reduce the number of temporary files.

Ksconf v0.12.0 (2023-09-27)

- Drop `lxml` and `endpoints` dependencies.
- Moved pre-commit hooks to ksconf-pre-commit repo, and started deprecation of the hooks in the main ksconf repo.

3.13.3 Ksconf 0.11

Highlights:

- Ksconf is beginning to treat Splunk apps more holistically and not just as a collection of `.conf` files.
- Significant portions of this new code base is directly leveraged by the Ansible modules located in the cdillc.splunk collection, a sibling project to Ksconf. some of the code code there has made it's way into the core ksconf project in this release.

API Changes

- Added `AppFacts` to easily collect Splunk application name, version, label, and other nuggets from `app.conf`.
- Added `AppManifest` to inventory the contents of a Splunk application and create a unique content fingerprint that can be used to quickly identify application changes.
- Added `ksconf.app.deploy` to assist with Splunk application deployment planning and execution.
- Added `ksconf.hookspec.KsconfHookSpecs` to define all available pluggy integration points. Anyone wanting to implement a new plugin should use the public-facing `ksconf.hook` module.

Ksconf v0.11.9 (2023-09-26)

- Splunk app packaging changes only. No need to update the package for CLI usage.
- Fix packaging bug impacting the Ksconf Splunk app. This issue was introduced in v0.11.6. Thanks to yohonet for brining this to my attention.
- Note that this took two release cycles to fully resolve. (Ignore ksconf v0.11.8)

Ksconf v0.11.7 (2023-09-20)

- Support disabling of plugins by name via `KSCONF_PLUGIN_DISABLE` environment variable. This expects a space separated lists of plugin names.
- Add new plugins documentation.

Ksconf v0.11.6 (2023-09-20)

- Introducing plugin functionality using pluggy plugin management system. This adds a small, single-package dependency that can greatly increase customization potential of ksconf. The first demo of this can be seen in the `ksconf-jinja-markdown` package that enables `.j2` payloads to be rendered by registering a custom Jinja filter named `markdown2html`.

Ksconf v0.11.5 (2023-08-25)

- Use atomic file operations for more updates (such as most `.conf` writing and app packaging). This is enabled by the new context managers `atomic_writer()` and `atomic_open()`. Under the covers temporary files are written to and then renamed into place to ensure that the output file is either fully updated or not touched at all. This should reduce possible data loss and/or confusion during some difficult to handle corner cases (i.e., disk full, or interrupted execution).
- Add new `parse_string()` function to enable simple parsing from a string. (I'm not sure why it took me so long to add this; it's so simple; and I've looked for this function dozens of times over the years, and always came up with a new StringIO workaround.)
- App Manifest changes:
  - App manifest file format was updated to v2 which reports file mode as as familiar octal (string) instead of an integer. This is easier to read in most cases.
  - Add API level improvements to support reading/writing manifests when the archive is using a temporary filename.
- Improved various error messages, minor optimizations, and some minor security improvements.

Ksconf v0.11.4 (2023-06-09)

- Updated Jinja2 support to allow variables to be passed in for the `combine` and `package` operations. To use Jinja2 rendering feature, use the `--enable-handler=jinja` option. Forcing users to op-in to this behavior seems to be safest option moving forward. To feed variables into the Jinja2 rendering process, use the `--template-vars` option. Either pass a literal value or `@filename`. Currently JSON literals as well as `.json`, `.yaml` and `.conf` files.
- Minor output fixes for `combine` (failed variable substitution)
- Rename LayerConfig to `LayerContext`. I doubt anyone is using this, but just in case.

Ksconf v0.11.3 (2023-05-17)

- Initial support for rendering Jinja2 templates in app layers, which can be used for app packaging and combine operations. This isn't really exposed via the CLI yet. Notice that if you have `*.j2` files in your apps that are NOT Jinja2 templates, this could cause problems for you. There's no way to disable this functionality as of yet.
- Move `.conf` and `.spec` combine handlers into handler functions. This makes it easier to supporting additional file types with special merge-handling logic in the future and simplifies the code into smaller units.
- More internal embracing of pathlib and typing.

Ksconf v0.11.0 (2023-05-13)

- First release of all `ksconf.app.*` functionality!
- The `unarchive` command has been updated to use the new functionality.
- More embracing of Python 3, f-strings, and pathlib!

NOTE: If you don't need for the new Splunk app functionality, there's little value in upgrading to v0.11. There's no new CLI features exposing this new functionality yet.

Disregard version v0.11.1, and v0.11.2 has minor internal fixes and shouldn't be broadly used. They were also released on 2023-05-13.

3.13.4 Ksconf 0.10

Highlights:

- Ksconf now requires Python 3.7 or newer.
- The Python package was renamed `ksconf`.

API Changes

- Core layer combining logic now lives in `LayerCombiner`. The new `RepeatableCombiner` class has logic for marker safety checks and settings for removing or preserving existing files. The `CombineCmd` now contains only the command line functionality.

Ksconf v0.10.2 (2023-05-13)

- Fix an `unarchive` bug triggered by trailing slashes in `--app-name`. Trailing slashes are removed automatically. Any other `/` present will trigger an error and require the user to clarify.

Ksconf v0.10.1 (2023-03-07)

- **Fix for pre-commit hook installation.** Pre-commit now uses an alternate shallow clone that fails to fetch the actual tag pointed to by `rev`. The ksconf python packaging process relies on git metadata. This results in an error message `InvalidVersion: Invalid version: 'cec3615'` in the pre-commit.log file. See pre-commit #2610 for additional background.

Ksconf v0.10.0 (2023-03-03)

Highlights:

- The official Python package was renamed `ksconf`. The `kintyre-splunk-conf` package will continue to be released in parallel at least until version 1.0. You can continue updating and using the `kintyre-splunk-conf` package but eventually startup warnings will be added to remind users to switch.
- Ksconf v0.10 and later requires Python 3.7 or newer. If you need Python 2.7 or 3.6 support, please stick with the latest 0.9.x release of `kintyre-splunk-conf`.
- The KSCONF acronym has taken on a new meaning. Originally, meaning *Kintyre's Splunk CONFiguration tool*, now becomes a recursive acronym: *Ksconf Splunk CONFiguration tool*. Kintyre has been acquired by CDI LLC, and this option seemed least intrusive.
- Add support Dashboard Studio dashboards. The JSON blobs inside of Simple XML payloads can now be formatted too. Multiline searches are still difficult to diff, but there's no way to fix that while using JSON.

More changes:

- Remove `six` built-in dependency.
- Refactor the `combine` logic into more reusable classes. This simplifies the CLI logic for both the `combine` and `package` functionality. The `combine` CLI and functionality remains unchanged.
- Updated `package` to use new combine new layer classes rather than making internal CLI calls to "combine". This has the potentially to be more efficient and allow for easier functionality expansions in the future.
- Deprecated the `--layer_method=auto` choice from the `package` command. This will be an error in the next release (v0.11).
- Add new layer-related dynamic variables for the `package` command. Use `{{layers_list}}` to capture what unique layer names made their way into a tarball, and use `{{layers_hash}}` when that list get too long to be manageable.

Bug fixes:

- Fixed `sort` bug where the user was incorrectly told that a file with errors was unexpectedly also successfully replaced. The contradictory output messages have been cleaned up. For clarity, this only occurred for inline replacement mode, and was purely a reporting issue, not a file handling problem.

- Fixed compatibility issues with `rest-publish` command and the splunk-sdk library around data type expectations. A big thanks to bayeslearner (#95) for the fix. If you run into any issues, try upgrading your version of `splunk-sdk`.

3.13.5 Ksconf 0.9

Highlights:

- Last version to support Python 2! It's time.

API Changes

- Removed `match_bwlist()` `FilteredList` and derived classes should be used instead.
- Updated interface for `compare_cfgs` and `compare_stanzas`. (1) Removed the `preserve_empty` parameter and (2) Replaced the awkwardly named `allow_level0` parameter with a new `replace_level` attribute that can be set to `global`, `stanza` or `key`. This new option can be used to control the level of detail in the output.

Ksconf v0.9.3 (2022-02-26)

- Added internal caching for `AppVarMagic` (`ksconf package` command) to reduce repeated variable expansion work. This will likely go unnoticed by most, but it does speed up some operations in the `cdillc.splunk.ksconf_package` Ansible module.
- Minor docs corrections.

Ksconf v0.9.2 (2022-03-04)

- The `filter` command can now include/exclude stanzas based on the boolean value of `disabled` using the new `--enabled-only` or `--disabled-only` arguments. The default behavior remains the same, that is, the `disabled` attribute is completely ignored. Thanks to John B Splunker for inspiring this feature!

Ksconf v0.9.1 (2022-03-03)

- Ksconf now tries harder to preserve file modification times. This is supported in `merge`, `combine` and `package` commands. Specifically, merged `.conf` files and concatenated files will keep the most recent modification time in the destination. This should make the output of `combine` and `package` (by extension) more deterministic in many scenarios.

Ksconf v0.9.0 (2021-08-12)

Features & Enhancements:

- Add new `--keep-existing` option for `ksconf combine` to preserve certain files that exist within the target directory but not within any source. Similarly the new `--disable-cleanup` option will prevent any files from being removed. This is useful, for example if using `ksconf combine` to write apps into `deployment-apps` where Splunk automatically creates a local `app.conf` file, and the deletion and recreation of the file can result in unnecessary app redeployments. These new options can be used together; for example, one useful pattern is to use `--disable-cleanup` to block all removals while perfecting/testing `--keep-existing` patterns.
- Add support for previewing stanza changes with `ksconf promote` by combining `--stanza X` and `--summary` options at the same time. Thanks to guilhemmarchand for the suggestion. (#89)
- New CLI args for `ksconf diff`. (1) New `--detail` option to specify how to handle certain 'replace' levels which impacts the way certain changes are represented. (2) New `--format json` for a more parsable output format. Note: This json format shouldn't be considered stable at this time. If you have ideas about how this could be used, please reach out.
- Allow enabling/disabling TTY colors via environmental variable. The new `--disable-color` option will disable color, or to disable more widely, add something like `export KSCONF_TTY_COLOR=off` to your bashrc profile or Windows environment variables.

Bug fixes:

- Fixed layer detection bugs for `dir.d` mode for layers. (1) Layers that weren't immediately under the source directory were not detected, and (2) layers existing beyond a symlink were not detected. This change targeted for `ksconf combine` but may fix other similar issues.
- Fixed #91. where `ksconf diff` wouldn't correctly handle empty stanzas in the second input file (Reversing the order would sometimes worked to avoid the issue). This was resolved by enabling some improved empty stanza handling in the conf comparison algorithms that were updated back in 0.7.10, but never globally applied. This has been resolved.

Documentation improvements

- New git tip: Use a `gitdir:` pointer to relocate the `.git` dir to avoid replicating it when a directory like `master-apps` is a git working copy.
- Additional quick use case in the cheatsheet page. Demonstrate how ksconf could be used to list all "apps" present on a deployment server from the `serverclass.conf` file.

API Change:

- Replaced use of `match_bwlist()` with the `FiltedListSplunkGlob` class, which allows old code to be cleaned up and technically, there's some expanded capabilities because of this (like many filters now supporting `file://filter.txt` type syntax, but this hasn't been documented and may be left as an Easter egg; because who reads changelogs?)
- Dropped `tty_color()` which had already been replaced with the `TermColor` class.

3.13.6 Ksconf 0.8

Highlights:

- New command *ksconf package* is designed for both Splunk developers and admins * New module `ksconf.builder` helps build Splunk apps using a pipeline; or when external Python libraries are bundled into an app
- Legit layer support with built-in layer filtering capabilities is available in several commands
- Python 3! Head's up: We'll be dropping support for Python 2 in an upcoming release

---

Note: Come chat about ksconf on GitHub discussions even if it's to say we should use some other forum to stay in touch.

What's new:

- The **new ksconf package command** supports the creation of Splunk app `.spl` files from a source directory. The `package` command can be used by admins to transfer apps around an organization, while keeping the `local` folder intact, or by a developer who wants `local` to be automatically merged into `default`. The app version can be set based on the latest git tag by simply saying `--set-version={{git_tag}}`.
- The **ksconf.builder Python module** is a API-only first for ksconf! This build library allow caching of expensive deterministic build operations, and has out-of-the-box support for frequent build steps like adding Python modules locally using `pip`. As the first feature with no CLI support, I'm exceeded to get input from the broader community on this approach. Of course this is just an experimental first release. As always, feedback welcome!
- **Native support for layers!** It's official, layers are now a proper ksconf feature, not just an abstract concept that you could throw together yourself given enough time and effort. This does mean that ksconf has to be more opinionated, but the design supports switching layer methods, which can be extended over time to support new different strategies as they emerge and are embraced by the community. Supports layers filtering as a native feature. This has always been technically possible, but awkward to implement yourself. Layer support is currently available in *ksconf combine* and *ksconf package* commands.
- **Moving to Python 3 soon.** In preparation for the move to Python 3, I've added additional backport libraries to be installed when running Python 2. Support for Python 2 will be dropped in a future release, and anyone still on Splunk 7 who can't get a Python 3 environment will have to use an older version of ksconf. Also note that when jumping to Python 3, we will likely be requiring Python 3.6 or newer right out of the gate. (This means dropping Python 2.7, 3.4 and 3.5 all at the same time.) Whoohoo for f-strings!
- **CLI option abbreviation has been disabled.** This could be a breaking change for existing scripts. Hopefully no one was relying on this already, but in order to prevent long-term CLI consistency issues as new CLI arguments are added, this feature has been disabled for all version of Python. This feature is only available, and was enabled by default, starting in Python 3.5.

- **Removed insensitive language.** Specifically the terms 'whitelist' and 'blacklist' have been replaced, where possible. Fortunately, these terms were not used in any CLI arguments, so there should be no user-facing changes as a result of this.
- **Removed support for building a standalone executable (zipapp).** This packaging option was added in v0.4.3, and deprecated in v0.6.0 once the Splunk app install option became available. I'm pretty sure this won't be missed.

API Changes

- NEW API `ksconf.builder` The documentation for this module needs work, and the whole API should be considered quite experimental. The easiest way to get started is to look at the *Build Example*.
- NEW Context manager `update_conf`. This enables super easy conf editing in Python with just a few lines of code. See docs API docs for a usage example.

Developer changes:

- Formatting via autopep8 and isort (enforced by pre-commit)
- Better flake8 integration for bulk checking (run via: `tox -e flake8,flake8-unittest`)

Ksconf v0.8.7 (2020-04-29)

- Support combining `*.conf.spec` files in `ksconf combine`, thus allowing `README.d` to be it's own layer.
- Fixed potential `unarchive` issue with older version of git where `git add --all DIR` is more explicit, but equivalent to the modern day, `git add DIR`.

Ksconf v0.8.6 (2020-04-20)

- Fixed `install.py` Splunk app CLI install helper script to support referencing a specific version of Python. This is needed on Splunk 8.0 if you'd like to use Python 3 (or Splunk 8.1 if you want to use Python 2.7, but please don't.) I suppose this would also work with using a custom Python interpreter other than the ones Splunk ships with, but then why not install with `pip`, right? (Thanks to guilhem.marchand for bringing this issue to my attention.)
- Updated docs regarding changes to the use of `install.py` and fixed a bunch of spelling mistakes and other minor doc/comment tweaks.
- Fixed ASCII art issue.

Ksconf v0.8.5 (2020-04-07)

- Fixed packaging issue where external dependencies were missing. This doesn't impact the Splunk package install, or anyone running Python 3.6 or later.

Ksconf v0.8.4 (2020-03-22)

- **CLI change**: Replaced short option for `--allowlist` to be `-a`, before it was `-w`. I assume this was left over early development where the argument was initial called `--whitelist`, but at this point `-w` is just confusing. Normally, I'd keep `-w` for a period of time and issue a deprecation warning. However, given that 0.8.0 was released less than a week ago, and that ksconf package is an "alpha" feature, I'm going to make this change without prior warning.
- Add some safety checks to the package command to check for app naming issues (where the app folder doesn't match `[package] id` value in `app.conf`), and hidden files and directories.
- Add new `{{app_id}}` variable that's usable with the ksconf package command.
- Added a new optional argument to `copy_files()` called `target` for additional control over the destination path of artifacts copied into the build folder.
- Minor tweak to unhandled exceptions. The name of the exception class is now show, and may be helpful in some situations.
- When using `make_missing` in `update_conf`, missing directories will now be created too.
- Additional fixes to the Ksconf for Splunk App `build.py` script: Now explicitly creating a top-level `ksconf` folder. It's likely that this was the root cause of several other issues.

Ksconf v0.8.3 (2021-03-20)

- Fixed bugs created by v0.8.2 (yanked on pypi)
- Properly resolved issues with Splunk app building process.
- Open issue uncovered where `ksconf package` can produce a tarball that's unusable by Splunkbase.

Ksconf v0.8.1 (2021-03-20)

- Fixed some build issues with the Splunk app. (The splunk app is now built with `ksconf package` and the `ksconf.builder`)
- Minor doc fix up; you know, the stuff typically found minutes after any new release :-)

Ksconf v0.8.0 (2021-03-19)

In addition to the 0.8 summary above, 0.8.0 specifically includes the following changes:

- Add automatic layer support. Currently the two supported layer schemes are (1) explicit layers (really this will `disable` automatic layer detection), and (2) the `dir.d` format which uses the `default.d/##-layer-name` style directory support, which we previously promoted in the docs, but never really *fully* supported in a native way. This new `dir.d` directory layout support also allows for multiple `*.d` folders in a single tree (so not just `default.d`), and if your apps have different layer-points in different apps, it's all handled transparently.
- Layer selection support was added to the `combine` command. This allows you to `--include` and `--exclude` layers as you see fit. See the docs for more details and examples of this new functionality. This works for both the new `dir.d` directories and the explicit layers, though moving to the `dir.d` format is highly encouraged.
- New cheatsheet example: Using `ksconf package` and `splunk install app` together.
- Updated the combine behavior to optimize for the situation where there is only a single conf input file provided. This behavior leaves any `.conf` or `.meta` file untouched so there's no sorting/normalizing or banner. See #64.
- Eliminated an "unknown command" error when one of the ksconf python modules has a SyntaxError. The new behavior isn't perfect (you may still see "unrecognized arguments"), but overall it's still a step in the right direction.

3.13.7 Ksconf 0.7.x

New functionality, massive documentation improvements, metadata support, and Splunk app install fixes.

Release v0.7.10 (2021-03-19)

- Fixed bug where empty stanzas in the local file could result in deletion in default with `ksconf promote`. Updated diff interface to improve handling of empty stanzas, but wider support is still needed across other commands; but this isn't a high priority.

Release v0.7.9 (2020-09-23)

- Fixed bug where empty stanzas could be removed from `.conf` files. This can be detrimental for `capability::*` entries in `authorize.conf`, for example. A big thanks to nebffa for tracking down this bug!

Release v0.7.8 (2020-06-19)

- New automatic `promote` mode is now available using CLI arguments! This allows stanzas to be selected for promotion from the CLI in batch and interactive modes. This implementation borrows (and shares code) with the `ksconf filter` command so hopefully the CLI arguments look familiar. It's possible to promote a single stanza, a stanza wildcard, regex or invert the matching logic and promote everything except for the named stanza (blocklist). Right now `--stanza` is the only supporting matching mode, but more can be added as needed. A huge thanks to mthambipillai for providing a pull-request with an initial implementation of this feature!
- Added a new summary output mode (`ksconf promote --summary`) that will provide a quick summary of what content could be promoted. This can be used along side the new `--stanza` filtering options to show the names of stanzas that can be promoted.
- Replaced insensitive terminology with race-neutral terms. Specifically the terms 'blacklist' and 'whitelist' have been replaced. NOTE: This does *not* change any CLI attributes, but in a few cases the standard output terminology is slightly different. Also terminology in `.conf` files couldn't be updated as that's controlled by Splunk.
- Fixed bug in the `unarchive` command where a `locale` folder was blocked as a `local` folder and where a nested `default` folder (nested under a Python package, for example) could get renamed if `--default-dir` was used, now only the top-most `default` folder is updated. Also fixed an unlikely bug triggered when `default/app.conf` is missing.
- Fixed bug with `minimize` when the required `--target` argument is not given. This now results in a reminder to the user rather than an unhandled exception.
- Splunk app packaging fix. Write access to the app was previously not granted due to a spelling mistake in the metadata file.

Release v0.7.7 (2020-03-05)

- Added new `--follow-symlink` option to the `combine` command so that input directory structures with symbolic links can be treated the same as proper directories.
- Corrected Windows issue where wildcard (glob) patterns weren't expanded by for `check` and `sort`. This is primarily a difference in how a proper shells (e.g., bash, csh, zsh) handle expansion natively vs CMD on Windows does not. However, since this is typically transparently handled by many CLI tools, we'll follow suite. (BTW, running ksconf from the GIT Bash prompt is a great alternative.) Only the most minimalistic expansion rules will be available, (so don't expect `{props,transforms,app}.conf` to work anytime soon), but this should be good enough for most use cases. Thanks to SID800 for reporting this bug.
- Fixed issues with the `unarchive` command when `git` is not installed or an app is being unarchived (installed/upgrade) into a location not managed by Git. Note that additional output is now enabled when the `KSCONF_DEBUG` environmental variable is set (in lieu of a proper verbose mode). Bug report provided by SID800.
- Enhanced `ksconf --version` output to include Git executable path and version information; as well as a platform dump. (Helpful for future bug reporting.)

- Added feature to disable the marker file (safety check) automatically created by the `combine` command for use in automated processing workflows.
- Updated `pre-commit` documentation and sample configurations to use `rev` rather than `sha` as the means of identifying upstream tags or revisions. Recent releases of `pre-commit` will warn you about this during each run.
- Fixed a temporary file cleanup issue during certain in-place file replacement operations. (If you found any unexpected `*.tmp` files, this could have been the cause.)

Release v0.7.6 (2019-08-15)

- Fresh review and cleanup of all docs! (A huge thank you to Brittany Barnett for this massive undertaking)
- Fixed unhandled exception when encountering a global stanza in metadata files.
- Expand some error messages, sanity checks, and added a new session token (`--session-key`) authentication option for `rest-publish`.

Release v0.7.5 (2019-07-03)

- Fixed a long-term bug where the diff output of a single-line attribute change was incorrectly represented in the textual output of 'ksconf diff' and the diff output in other commands. This resolves a combination of bugs, the first half of which was fixed in 0.7.3.
- Allow `make_docs` script to run on Windows, and other internal doc build process improvements.

Release v0.7.4 (2019-06-07)

- Inline the `six` module to avoid elusive bootstrapping cases where the module couldn't be found. This primarily impacts `pre-commit` users. The `ksconf.ext.*` prefix is being used for this, and any other inlined third party modules we may need in the future.
- Other minor docs fixes and internal non-visible changes.

Release v0.7.3 (2019-06-05)

- Added the new *ksconf xml-format* command.
  - The `ksconf xml-format` command brings format consistency to your XML representations of Simple XML dashboards and navigation files by fixing indentation automatically adding `<![CDATA[ ... ]]>` blocks, as needed, to reduce the need for XML escaping, resulting in more readable source.
  - Additionally, a new pre-commit hook named *ksconf-xml-format* was added to leverage this new functionality. It looks specifically for xml views and navigation files based on

path. This may also include Advanced XML, which hasn't been tested; So if you use Advanced XML, proceed with caution.

  - Note that this adds `lxml` as a packaging dependency which is needed for pre-commit hooks, but not strictly required at run time for other ksconf commands. This is NOT ideal, and may change in the future in attempts to keep ksconf as light-weight and standalone as possible. One possible alternative is setting up a different repo for pre-commit hooks. Python packaging and distribution tips welcome.

- Fixed data loss bug in `promote` (interactive mode only) and improved some UI text and prompts.
- Fixed colorization of `ksconf diff` output where certain lines failed to show up in the correct color.
- Fixed bug where debug tracebacks didn't work correctly on Python 2.7. (Enable using `KSCONF_DEBUG=1`.)
- Extended the output of `ksconf --version` to show the names and version of external modules, when present.
- Improved some resource allocation in corner cases.
- Tested with Splunk 7.3 (numeric similarity in version numbers is purely coincidental)

Attention: API BREAKAGE

The `DiffOp` output values for `DIFF_OP_INSERT` and `DIFF_OP_DELETE` have been changed in a backwards-compatible breaking way. The values of `a` and `b` were previously reversed for these two operations, leading to some code confusion.

Release v0.7.2 (2019-03-22)

- Fixed bug where `filter` would crash when doing stanza matching if global entries were present. Global stanzas can be matched by searching for a stanza named `default`.
- Fixed broken `pre-commit` issue that occurred for the `v0.7.1` tag. This also kept `setup.py` from working if the `six` module wasn't already installed. Developers and pre-commit users were impacted.

Release v0.7.1 (2019-03-13)

- Additional fixes for UTF-8 BOM files which appear to happen more frequently with `local` files on Windows. This time some additional unit tests were added so hopefully there are few regressions in the future.
- Add the `ignore-missing` argument to *ksconf merge* to prevent errors when input files are absent. This allows bashisms `Some_App/{{default,local}}/savedsearches.conf` to work without errors if the local or default file is missing.

- Check for incorrect environment setup and suggest running sourcing `setSplunkEnv` to get a working environment. See #48 for more info.
- Minor improvements to some internal error handling, packaging, docs, and troubleshooting code.

Release v0.7.0 (2019-02-27)

> **Attention:** For anyone who installed 0.6.x, we recommend a fresh install of the Splunk app due to packaging changes. This shouldn't be an issue in the future.

General changes:

- Added new *ksconf rest-publish* command that supersedes the use of `rest-export` for nearly every use case. Warning: No unit-testing has been created for this command yet, due to technical hurdles.
- Added *Cheat Sheet* to the docs.
- Massive doc cleanup of hundreds of typos and many expanded/clarified sections.
- Significant improvement to entrypoint handling and support for conditional inclusion of 3rd party libraries with sane behavior on import errors, and improved warnings. This information is conveniently viewable to the user via `ksconf --version`.
- Refactored internal diff logic and added additional safeties and unit tests. This includes improvements to TTY colorization which should avoid previous color leaks scenarios that were likely if unhandled exceptions occur.
- New support for metadata handling.
- CLI change for `rest-export`: The `--user` argument has been replaced with `--owner` to keep clean separation between the login account and object owners. (The old argument is still accept for now.)

Splunk app changes:

- Modified installation of python package installation. In previous releases, various `.dist-info` folders were created with version-specific names leading to a mismatch of package versions after upgrade. For this reason, we suggest that anyone who previously installed 0.6.x should do a fresh install.
- Changed Splunk app install script to `install.py` (it was `bootstrap_bin.py`). Hopefully this is more intuitive.
- Added Windows support to `install.py`.
- Now includes the Splunk Python SDK. Currently used for `rest-publish` but will eventually be used for additional functionally unique to the Splunk app.

3.13.8 Ksconf 0.6.x

Add deployment as a Splunk app for simplicity and significant docs cleanup.

Release v0.6.2 (2019-02-09)

- Massive rewrite and restructuring of the docs. Highlights include:
  - Reference material has been moved out of the user manual into a different top-level section.
  - Many new topics were added, such as
    - *Ksconf as external difftool*
    - *How Splunk writes to conf files*
    - *Configuration layers*
    - *What's so important about minimizing files?*
  - A new approach for CLI documentation. We're moving away from the **WALL OF TEXT** thing. (Yeah, it was really just the output from `--help`). That was limiting formatting, linking, and making the CLI output way too long.
- Refreshed Splunk app icons. Add missing alt icon.
- Several minor internal cleanups. Specifically the output of `--version` had a face lift.

Release v0.6.1 (2019-02-07)

- (Trivial) Fixed some small issues with the Splunk App (online AppInspect)

Release v0.6.0 (2019-02-06)

- Add initial support for building ksconf into a Splunk app.
  - App contains a local copy of the docs, helpful for anyone who's working offline.
  - Credit to Sarah Larson for the ksconf logos.
  - No `ksconf` functionality exposed to the Splunk UI at the moment.
- Docs/Sphinx improvements (more coming)
  - Begin work on cleaning up API docs.
  - Started converting various document pages into reStructuredText for greatly improved docs.
  - Improved PDF fonts and fixed a bunch of sphinx errors/warnings.

- Refactored the install docs into 2 parts. With the new ability to install ksconf as a Splunk app it's quite likely that most of the wonky corner cases will be less frequently needed, hence all the more exotic content was moved into the "Advanced Install Guide", tidying things up.

3.13.9 Ksconf 0.5.x

Add Python 3 support, new commands, support for external command plugins, tox and vagrant for testing.

Release v0.5.6 (2019-02-04)

- Fixes and improvements to the `filter` command. Found issue with processing from stdin, inconsistency in some CLI arguments, and finished implementation for various output modes.
- Add logo (fist attempt).

Release v0.5.5 (2019-01-28)

- New *ksconf filter* command added for slicing up a conf file into smaller pieces. Think of this as GREP that's stanza-aware. Can also allow or block attributes, if desirable.
- Expanded `rest-export` CLI capabilities to include a new `--delete` option, pretty-printing, and now supports stdin by allowing the user to explicitly set the file type using `--conf`.
- Refactored all CLI unittests for increased readability and long-term maintenance. Unit tests now can also be run individually as scripts from the command line.
- Minor tweaks to the `snapshot` output format, v0.2. This feature is still highly experimental.

Release v0.5.4 (2019-01-04)

- New commands added:
  - *ksconf snapshot* will dump a set of configuration files to a JSON formatted file. This can be used used for incremental "snapshotting" of running Splunk apps to track changes overtime.
  - *ksconf rest-export* builds a series of custom `curl` commands that can be used to publish or update stanzas on a remote instance without file system access. This can be helpful when pushing configs to Splunk Cloud when all you have is REST (splunkd) access. This command is indented for interactive admin not batch operations.
- Added the concept of command maturity. A listing is available by running `ksconf --version`
- Fix typo in `KSCONF_DEBUG`.
- Resolving some build issues.
- Improved support for development/testing environments using Vagrant (fixes) and Docker (new). Thanks to Lars Jonsson for these enhancements.

Release v0.5.3 (2018-11-02)

- Fixed bug where `ksconf combine` could incorrectly order directories on certain file systems (like ext4), effectively ignoring priorities. Repeated runs may resulted in undefined behavior. Solved by explicitly sorting input paths forcing processing to be done in lexicographical order.
- Fixed more issues with handling files with BOM encodings. BOMs and encodings in general are NOT preserved by ksconf. If this is an issue for you, please add an enhancement issue.
- Add Python 3.7 support
- Expand install docs specifically for offline mode and some OS-specific notes.
- Enable additional tracebacks for CLI debugging by setting `KSCONF_DEBUG=1` in the environment.

Release v0.5.2 (2018-08-13)

- Expand CLI output for `--help` and `--version`
- Internal cleanup of CLI entry point module name. Now the ksconf CLI can be invoked as `python -m ksconf`, you know, for anyone who's into that sort of thing.
- Minor docs and CI/testing improvements.

Release v0.5.1 (2018-06-28)

- Support external ksconf command plugins through custom *entry_points*, allowing for others to develop their own custom extensions as needed.
- Many internal changes: Refactoring of all CLI commands to use new entry_points as well as pave the way for future CLI unittest improvements.
- Docs cleanup / improvements.

Release v0.5.0 (2018-06-26)

- Python 3 support.
- Many bug fixes and improvements resulting from wider testing.

3.13.10 Ksconf 0.4.x

Ksconf 0.4.x switched to a modular code base, added build/release automation, PyPI package registration (installation via `pip install` and, online docs.

Release v0.4.10 (2018-06-26)

- Improve file handling to avoid "unclosed file" warnings. Impacted `parse_conf()`, `write_conf()`, and many unittest helpers.
- Update badges to report on the master branch only. (No need to highlight failures on feature or bug-fix branches.)

Release v0.4.9 (2018-06-05)

- Add some missing docs files

Release v0.4.8 (2018-06-05)

- Massive cleanup of docs: revamped install guide, added 'standalone' install procedure and developer-focused docs. Updated license handling.
- Updated docs configuration to dynamically pull in the ksconf version number.
- Using the classic 'read-the-docs' Sphinx theme.
- Added additional PyPi badges to README (GitHub home page).

Release v0.4.4-v0.4.7 (2018-06-04)

- Deployment and install fixes (It's difficult to troubleshoot/test without making a new release!)

Release v0.4.3 (2018-06-04)

- Rename PyPI package `kintyre-splunk-conf`
- Add support for building a standalone executable (zipapp).
- Revamp install docs and location
- Add GitHub release for the standalone executable.

Release v0.4.2 (2018-06-04)

- Add readthedocs.io support

Release v0.4.1 (2018-06-04)

- Enable PyPI production package building

Release v0.4.0 (2018-05-19)

- Refactor entire code base. Switched from monolithic all-in-one file to clean-cut modules.
- Versioning is now discoverable via `ksconf --version`, and controlled via git tags (via `git describe --tags`).

Module layout

- `ksconf.conf.*` - Configuration file parsing, writing, comparing, and so on
- `ksconf.util.*` - Various helper functions
- `ksconf.archive` - Support for decompressing Splunk apps (tgz/zip files)
- `ksconf.vc.git` - Version control support. Git is the only VC tool supported for now. (Possibly ever)
- `ksconf.commands.<CMD>` - Modules for specific CLI functions. I may make this extendable, eventually.

3.13.11 Ksconf 0.3.x

First public releases.

Release v0.3.2 (2018-04-24)

- Add AppVeyor for Windows platform testing
- Add codecov integration
- Created ConfFileProxy.dump()

Release v0.3.1 (2018-04-21)

- Setup automation via Travis CI
- Add code coverage

Release v0.3.0 (2018-04-21)

- Switched to semantic versioning.
- 0.3.0 feels representative of the code maturity.

3.13.12 Ksconf legacy releases

Ksconf started in a private Kintyre repo. There are no official releases; all git history has been rewritten.

Release legacy-v1.0.1 (2018-04-20)

- Fixes to blocklist support and many enhancements to `ksconf unarchive`.
- Introduces parsing profiles.
- Lots of bug fixes to various subcommands.
- Added automatic detection of 'subcommands' for CLI documentation helper script.

Release legacy-v1.0.0 (2018-04-16)

- This is the first public release. First work began Nov 2017 (as a simple conf 'sort' tool, which was imported from yet another repo.) Version history was extracted/rewritten/preserved as much as possible.
- Mostly stable features.
- Unit test coverage over 85%
- Includes pre-commit hook configuration (so that other repos can use this to run `ksconf sort` and `ksconf check` against their conf files.

3.14 Known issues

3.14.1 General

- File encoding issues: Byte order markers and specific encodings are NOT preserved. All files are encoding using UTF-8 upon update, which is Splunk's expected encoding.

3.14.2 Splunk app

- File cleanup issues after *KSCONF app for Splunk* upgrades (impacts versions prior to 0.7.0). Old `.dist-info` folders or other stale files may be left around after upgrades. If you encounter this issue, either uninstall and delete the ksconf directory or manually remove the old 'bin' folder and (re)upgrade to the latest version. The fix in 0.7.0 is to remove the version-specific portion of the folder name. (GH issue #37)

See more confirmed bugs in the issue tracker.

3.15 Advanced Installation Guide

The content in this document is a subsidiary to the *Installation Guide* because it became disorganized and the number of possible Python installation combinations and snags intensified. However, that culminated in the collection of excellent information that is provided here. Please remember, the Splunk app install approach was introduced to alleviate several of these issues.

A portion of this document is targeted at those who can't install packages as Admin or are forced to use Splunk's embedded Python. For everyone else, please start with the one-liner:

```
pip install -U ksconf
```

This document includes some legacy information that may not longer be true. Generally speaking, installing Python packages has become much easier since Python 2 went away. However, there are still some weird corner cases out there so this document has be kept around for reference.

Tip: Do any of these words for phrases strike fear in your heart?

- `pip`
- `pipenv`
- `virtualenv`
- `wheel`
- `pyenv` (not the same as `pyvenv`)
- `python3.7` vs `python37` vs `py -37`
- `PYTHONPATH`
- `LD_LIBARY`
- RedHat Software Collections

If this list seems daunting, head over to *Install Splunk App*. There's no shame in it.

Contents

3.15.1 Flowchart

(Unfinished; more of a brainstorm at this point. . .)

- Is Python installed? (OS level)
  - Is the version greater than 3.7?
- Do you have admin access? (root/Administrator; or can you get it? How hard? Will you need it each time you upgrade the ksconf?)
- Do you already have a large Python deployment or dependency? (If so, you'll probably be fine. Use venv)

- Do you have any prior Python packaging or administration experience?
- Are you dealing with some vendor-specific solution?
  - Example: RedHat Software Collections – where they realize their software is way too old, so they try to make it possible to install newer version of things like Python, but since they aren't native or the default, you still end up jumping through a bunch of wonky hoops)
- Do you have Internet connectivity? (air gap or blocked outbound traffic, or proxy)
- Do you want to build/deploy your own ksconf extensions? If so, the Python package is a better option. (But at that point, you can probably already handle any packaging issues yourself.)

3.15.2 Installation

There are several ways to install ksconf. Technically, all standard Python packaging approaches should work just fine. However, for non-Python developers, there are some snags. Installation options are listed from the most easy and recommended, to more obscure and difficult:

Install from PyPI with PIP

The preferred installation method is to install via the standard Python package tool **`pip`**. Ksconf can be installed via the registered ksconf package using the standard Python process.

There are 2 popular variations, depending on whether or not you would like to install for all users or test it locally.

Install ksconf into a virtual environment

Use this option if you don't have admin access

Installing `ksconf` with venv is a great way to test the tool without requiring admin privileges and has many advantages for a production install. Here are the basic steps to get started.

Note: Virtualenv vs venv

We used to recommend using virtualenv, which worked with Python 2 and 3. But since Python now ships with venv, there's no functional differences between the two approaches, we now suggest using 'venv'. That being said, virtualenv still works fine and will continue to be supported.

Please change `venv` to a suitable path for your environment.

```
# Create and activte new 'venv' virtual environment
python3 -m venv venv
source venv/bin/activate
```

(continues on next page)

(continued from previous page)

```
pip install ksconf
```

Note: Windows users

The above virtual environment activation should be run as `venv\Scripts\activate.bat`.

Install ksconf system-wide

Important: This requires admin access.

This is the absolute easiest install method where 'ksconf' is available to all users on the system but it requires root access and `pip` must be installed and up-to-date.

On Mac or Linux, run:

```
sudo pip install ksconf
```

On Windows, run this command from an Administrator console.

```
pip install ksconf
```

CentOS (RedHat derived) distros

```
# Enable the EPEL repo so that `pip` can be installed.
sudo yum install -y epel-release

# Install pip
sudo yum install -y python-pip

# Install ksconf (globally, for all users)
sudo pip install ksconf
```

RedHat Software Collections

The following assumes the `python38` software collection, but other version of Python are supported too. The initial setup and deployment of Software Collections is beyond the scope of this doc.

```
sudo scl enable python38 python -m pip install ksconf
```

Hint: Missing pip?

If pip is missing from a RHSC, then install the following rpm.

```
yum install python38-python-pip
```

Unfortunately, the `ksconf` entrypoint script (in the `bin` folder) will not work correctly on it's own because it doesn't know about the scl environment, nor is it in the default PATH. To solve this, run the following:

```
sudo cat > /usr/local/bin/ksconf <<HERE
#!/bin/sh
source scl_source enable python27
exec /opt/rh/python27/root/usr/bin/ksconf "$@"
HERE
chmod +x /usr/local/bin/ksconf
```

On Linux or Mac

Download the latest ksconf wheel file from PyPI. The path to this download will be set in the `pkg` variable as shown below.

Setup the shell:

```
export SPLUNK_HOME=/opt/splunk
export pkg=~/Downloads/kintyre_splunk_conf-0.4.9-py2.py3-none-any.whl
```

Run the following:

```
cd $SPLUNK_HOME
mkdir Kintyre
cd Kintyre
# Unzip the 'kconf' folder into SPLUNK_HOME/Kintyre
unzip "$pkg"

cat > $SPLUNK_HOME/bin/ksconf <<HERE
#!/bin/sh
export PYTHONPATH=$PYTHONPATH:$SPLUNK_HOME/Kintyre
```

(continues on next page)

(continued from previous page)

```
exec $SPLUNK_HOME/bin/python -m ksconf \$*
HERE
chmod +x $SPLUNK_HOME/bin/ksconf
```

Test the install:

```
ksconf --version
```

On Windows

1. Open a browser and download the latest ksconf wheel file from PyPI.
2. Rename the `.whl` extension to `.zip`. (This may require showing file extensions in Explorer.)
3. Extract the zip file to a temporary folder. (This should create a folder named "ksconf")
4. Create a new folder called "Kintyre" under the Splunk installation path (aka `SPLUNK_HOME`) By default, this is `C:\Program Files\Splunk`.
5. Copy the "ksconf" folder to `%SPLUNK_HOME%\Kintyre`.
6. Create a new batch file called `ksconf.bat` and paste in the following. Be sure to adjust for a non-standard `%SPLUNK_HOME%` value, if necessary.

   ```
   @echo off
   SET SPLUNK_HOME=C:\Program Files\Splunk
   SET PYTHONPATH=%SPLUNK_HOME%\bin;%SPLUNK_HOME%\Python-3.7\Lib\site-packages\
   ↪win32;%SPLUNK_HOME%\Python-3.7\Lib\site-packages;%SPLUNK_HOME%\Python-3.7\Lib
   SET PYTHONPATH=%PYTHONPATH%;%SPLUNK_HOME%\Kintyre
   CALL "%SPLUNK_HOME%\bin\python.exe" -m ksconf %*
   ```

7. Move `ksconf.bat` to the `Splunk\bin` folder. (This assumes that `%SPLUNK_HOME%/bin` is part of your `%PATH%`. If not, add it, or find an appropriate install location.)
8. Test this by running `ksconf --version` from the command line.

3.15.3 Offline installation

Installing ksconf to an offline or network restricted computer requires three steps: (1) download the latest packages from the Internet to a staging location, (2) transfer the staged content (often as a zip file) to the restricted host, and (3) use pip to install packages from the staged copy. Fortunately, pip makes offline workflows quite easy to achieve. Pip can download a Python package with all dependencies stored as wheels files into a single directory, and pip can be told to install from that directory instead of attempting to talk to the Internet.

The process of transferring these files is very organization-specific. The example below shows the creation of a tarball (since `tar` is universally available on Unix systems), but any acceptable

method is fine. If security is a high concern, this step is frequently where safety checks are implemented: such as, antivirus scans, static code analysis, manual inspection, and/or comparison of cryptographic file hashes.

One additional use-case for this workflow, is to ensure the exact same version of all packages are deployed consistently across all servers and environments. Often, building a `requirements.txt` file with `pip freeze`, is a more appropriate solution. Alternatively, consider using `pipenv lock` for even more security benefits.

Offline installation steps

Important: Pip must be installed on the destination server for this process to work. If pip is NOT installed, see the *Offline installation of pip* section below.

Step 1: Use pip to download the latest package and their dependencies. Be sure to use the same version of Python that is running on destination machine.

```
# download packages
python3 -m pip download -d ksconf-packages ksconf
```

A new directory named 'ksconf-packages' will be created and will contain the necessary `*.whl` files.

Step 2: Transfer the directory or archive to the remote computer. Insert whatever security and file copy procedures necessary for your organization.

```
# Compress directory (on staging computer)
tar -czvf ksconf-packages.tgz ksconf-packages

# Copy file using whatever means (for example, scp)
scp ksconf-packages.tgz user@server:/tmp/ksconf-packages.tgz

# Extract the archive (on destination server)
tar -xzvf ksconf-packages.tgz
```

Step 3:

```
# Install ksconf package with pip
pip install --no-index --find-links=ksconf-packages ksconf

# Test the installation
ksconf --version
```

The `ksconf-packages` folder can now be safely removed.

Offline installation of pip

Use the recommended `pip` install procedures listed elsewhere if possible. But if a remote bootstrap of pip is your only option, then here are the steps. (This process mirrors the steps above and can be combined, if needed.)

Step 1: Fetch bootstrap script and necessary wheels

```
mkdir ksconf-packages
curl https://bootstrap.pypa.io/get-pip.py -o ksconf-packages/get-pip.py
python3 -m pip download -d /tmp/my_packages pip setuptools wheel
```

The `ksconf-packages` folder should contain 1 script, and 3 wheel (`*.whl`) files.

Step 2: Archive and/or copy to offline server

Step 3: Bootstrap pip

```
sudo python get-pip.py --no-index --find-links=ksconf-packages/

# Test with
pip --version
```

Use pip without installing it

If you have a copy of the `pip*.whl` (wheel) file, then it can be executed directly by Python. This can be used to run `pip` without actually installing it, or for installing pip initially (bypassing the `get-pip.py` script step noted above.)

Here's an example of how this could work:

Step 1: Download the pip wheel on a machine where `pip` works, by running:

```
pip download pip -d .
```

This will create a file like `pip-19.0.1-py2.py3-none-any.whl` in the current working directory.

Step 2: Copy the pip wheel to another machine (likely where pip isn't installed.)

Step 3: Execute the wheel by running:

```
python pip-19.0.1-py2.py3-none-any.whl/pip list
```

Substitute the `list` command with whatever action you need (like `install` or whatever).

3.15.4 Frequent gotchas

PIP Install TLS Error

If `pip` throws an error message like the following:

```
There was a problem confirming the ssl certificate: [SSL: TLSV1_ALERT_PROTOCOL_
↪VERSION] tlsv1 alert protocol version
...
No matching distribution found for setuptools
```

The problem is likely caused by changes to PyPI website in April 2018 when support for TLS v1.0 and 1.1 were removed. Downloading new packages requires upgrading to a new version of pip. Like so:

Upgrade pip as follows:

```
curl https://bootstrap.pypa.io/get-pip.py | python
```

Note: Use `sudo python` above if not in a virtual environment.

Helpful links:

- Not able to install Python packages [SSL: TLSV1_ALERT_PROTOCOL_VERSION]
- 'pip install' fails for every package ("Could not find a version that satisfies the requirement")

No module named 'command.install'

If, while trying to install `pip` or run a `pip` command you see the following error:

```
ImportError: No module named command.install
```

Likely this is because you are using a crippled version of Python; like the one that ships with Splunk. This won't work. Either install the Splunk app package from Splunkbase or install using the OS-level Python.

3.15.5 Troubleshooting

Here are a few fact gathering type commands that may help you begin to track down problems.

Check Python version

Check your installed Python version by running:

```
python --version
```

Note that Linux distributions and Mac OS X that ship with multiple versions of Python may have renamed this to `python3`, `python3.8` or similar.

Check PIP Version

```
pip --version
```

If you are running a different Python interpreter version, you can instead run this as:

```
python3 -m pip --version
```

Validate the install

Confirm installation with the following command:

```
ksconf --version
```

If this works, it means that `ksconf` installed and is part of your `PATH` and should be useable everywhere in your system. Go forth and conquer!

If this doesn't work, here are a few things to try:

1. Check that your `PATH` is set correctly.
2. Try running ksconf as a "module" (sometimes works around a PATH issue). Run `python -m ksconf`
3. If you're running the Splunk app, try running the following:

   ```
   cd $SPLUNK_HOME/etc/apps/ksconf/bin/lib
   python -m ksconf --version
   ```

   If this works, then the issue is with PATH.

It may be helpful to uninstall (remove) the Splunk app and reinstall from scratch.

3.15.6 Resources

- Python packaging docs provide a general overview on installing Python packages, how to install per-user vs install system-wide.
- Install PIP docs explain how to bootstrap or upgrade `pip` the Python packaging tool. Python 3 comes with this by default, but some Linux distros break this into a separate package.

3.16 License

```
                                 Apache License
                           Version 2.0, January 2004
                        http://www.apache.org/licenses/

   TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

   1. Definitions.

      "License" shall mean the terms and conditions for use, reproduction,
      and distribution as defined by Sections 1 through 9 of this document.

      "Licensor" shall mean the copyright owner or entity authorized by
      the copyright owner that is granting the License.

      "Legal Entity" shall mean the union of the acting entity and all
      other entities that control, are controlled by, or are under common
      control with that entity. For the purposes of this definition,
      "control" means (i) the power, direct or indirect, to cause the
      direction or management of such entity, whether by contract or
      otherwise, or (ii) ownership of fifty percent (50%) or more of the
      outstanding shares, or (iii) beneficial ownership of such entity.

      "You" (or "Your") shall mean an individual or Legal Entity
      exercising permissions granted by this License.

      "Source" form shall mean the preferred form for making modifications,
      including but not limited to software source code, documentation
      source, and configuration files.

      "Object" form shall mean any form resulting from mechanical
      transformation or translation of a Source form, including but
      not limited to compiled object code, generated documentation,
      and conversions to other media types.

      "Work" shall mean the work of authorship, whether in Source or
      Object form, made available under the License, as indicated by a
```

(continues on next page)

(continued from previous page)

```
    copyright notice that is included in or attached to the work
    (an example is provided in the Appendix below).

    "Derivative Works" shall mean any work, whether in Source or Object
    form, that is based on (or derived from) the Work and for which the
    editorial revisions, annotations, elaborations, or other modifications
    represent, as a whole, an original work of authorship. For the purposes
    of this License, Derivative Works shall not include works that remain
    separable from, or merely link (or bind by name) to the interfaces of,
    the Work and Derivative Works thereof.

    "Contribution" shall mean any work of authorship, including
    the original version of the Work and any modifications or additions
    to that Work or Derivative Works thereof, that is intentionally
    submitted to Licensor for inclusion in the Work by the copyright owner
    or by an individual or Legal Entity authorized to submit on behalf of
    the copyright owner. For the purposes of this definition, "submitted"
    means any form of electronic, verbal, or written communication sent
    to the Licensor or its representatives, including but not limited to
    communication on electronic mailing lists, source code control systems,
    and issue tracking systems that are managed by, or on behalf of, the
    Licensor for the purpose of discussing and improving the Work, but
    excluding communication that is conspicuously marked or otherwise
    designated in writing by the copyright owner as "Not a Contribution."

    "Contributor" shall mean Licensor and any individual or Legal Entity
    on behalf of whom a Contribution has been received by Licensor and
    subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of
    this License, each Contributor hereby grants to You a perpetual,
    worldwide, non-exclusive, no-charge, royalty-free, irrevocable
    copyright license to reproduce, prepare Derivative Works of,
    publicly display, publicly perform, sublicense, and distribute the
    Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of
    this License, each Contributor hereby grants to You a perpetual,
    worldwide, non-exclusive, no-charge, royalty-free, irrevocable
    (except as stated in this section) patent license to make, have made,
    use, offer to sell, sell, import, and otherwise transfer the Work,
    where such license applies only to those patent claims licensable
    by such Contributor that are necessarily infringed by their
    Contribution(s) alone or by combination of their Contribution(s)
    with the Work to which such Contribution(s) was submitted. If You
    institute patent litigation against any entity (including a
```

(continues on next page)

(continued from previous page)

```
   cross-claim or counterclaim in a lawsuit) alleging that the Work
   or a Contribution incorporated within the Work constitutes direct
   or contributory patent infringement, then any patent licenses
   granted to You under this License for that Work shall terminate
   as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the
   Work or Derivative Works thereof in any medium, with or without
   modifications, and in Source or Object form, provided that You
   meet the following conditions:

   (a) You must give any other recipients of the Work or
       Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices
       stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works
       that You distribute, all copyright, patent, trademark, and
       attribution notices from the Source form of the Work,
       excluding those notices that do not pertain to any part of
       the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its
       distribution, then any Derivative Works that You distribute must
       include a readable copy of the attribution notices contained
       within such NOTICE file, excluding those notices that do not
       pertain to any part of the Derivative Works, in at least one
       of the following places: within a NOTICE text file distributed
       as part of the Derivative Works; within the Source form or
       documentation, if provided along with the Derivative Works; or,
       within a display generated by the Derivative Works, if and
       wherever such third-party notices normally appear. The contents
       of the NOTICE file are for informational purposes only and
       do not modify the License. You may add Your own attribution
       notices within Derivative Works that You distribute, alongside
       or as an addendum to the NOTICE text from the Work, provided
       that such additional attribution notices cannot be construed
       as modifying the License.

   You may add Your own copyright statement to Your modifications and
   may provide additional or different license terms and conditions
   for use, reproduction, or distribution of Your modifications, or
   for any such Derivative Works as a whole, provided Your use,
   reproduction, and distribution of the Work otherwise complies with
   the conditions stated in this License.
```

(continues on next page)

(continued from previous page)

```
   5. Submission of Contributions. Unless You explicitly state otherwise,
      any Contribution intentionally submitted for inclusion in the Work
      by You to the Licensor shall be under the terms and conditions of
      this License, without any additional terms or conditions.
      Notwithstanding the above, nothing herein shall supersede or modify
      the terms of any separate license agreement you may have executed
      with Licensor regarding such Contributions.

   6. Trademarks. This License does not grant permission to use the trade
      names, trademarks, service marks, or product names of the Licensor,
      except as required for reasonable and customary use in describing the
      origin of the Work and reproducing the content of the NOTICE file.

   7. Disclaimer of Warranty. Unless required by applicable law or
      agreed to in writing, Licensor provides the Work (and each
      Contributor provides its Contributions) on an "AS IS" BASIS,
      WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
      implied, including, without limitation, any warranties or conditions
      of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
      PARTICULAR PURPOSE. You are solely responsible for determining the
      appropriateness of using or redistributing the Work and assume any
      risks associated with Your exercise of permissions under this License.

   8. Limitation of Liability. In no event and under no legal theory,
      whether in tort (including negligence), contract, or otherwise,
      unless required by applicable law (such as deliberate and grossly
      negligent acts) or agreed to in writing, shall any Contributor be
      liable to You for damages, including any direct, indirect, special,
      incidental, or consequential damages of any character arising as a
      result of this License or out of the use or inability to use the
      Work (including but not limited to damages for loss of goodwill,
      work stoppage, computer failure or malfunction, or any and all
      other commercial damages or losses), even if such Contributor
      has been advised of the possibility of such damages.

   9. Accepting Warranty or Additional Liability. While redistributing
      the Work or Derivative Works thereof, You may choose to offer,
      and charge a fee for, acceptance of support, warranty, indemnity,
      or other liability obligations and/or rights consistent with this
      License. However, in accepting such obligations, You may act only
      on Your own behalf and on Your sole responsibility, not on behalf
      of any other Contributor, and only if You agree to indemnify,
      defend, and hold each Contributor harmless for any liability
      incurred by, or claims asserted against, such Contributor by reason
      of your accepting any such warranty or additional liability.
```

(continues on next page)

(continued from previous page)

```
END OF TERMS AND CONDITIONS

Copyright 2019 Kintyre

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
```

3.17 API Reference

Note: As of now, no assumptions should be made about APIs remaining stable

KSCONF is first and foremost a CLI tool, so backwards incompatible changes are more of a concern for CLI breakage than for API breakage. That being said, there are a number of helpful features in the core `ksconf` Python module. So if anyone is interested in using the API, please feel free to do so, but let us know *how* you are using it and we'll find a way to keep the the important bits stable. We'd love to make it more clear what APIs are stable and which are likely to change.

As of right now, the general rule of thumb is this: Anything well-covered by the unit tests should be be fairly safe to build on top of, but again, *ping us*.

3.17.1 KSCONF API

ksconf

ksconf namespace

Subpackages

ksconf.app package

Submodules

ksconf.app.deploy module

class ksconf.app.deploy.**DeployAction**(*action: 'str'*)

Bases: object

action: str

classmethod from_dict(*data: dict*) → *DeployAction*

to_dict() → dict

class ksconf.app.deploy.**DeployActionType**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

EXTRACT_FILE = 'extract_file'

REMOVE_FILE = 'remove'

Implement in future phase SET_SYMLINK = “link” UPDATE_META = “meta”

SET_APP_NAME = 'app'

SOURCE_REFERENCE = 'source'

class ksconf.app.deploy.**DeployAction_ExtractFile**(*subtype: 'str'*, *path: 'PurePosixPath'*, *mode: 'int' = None*, *mtime: 'int' = None*, *hash: 'str' = None*, *rel_path: 'str' = None*)

Bases: DeployAction

action: str = 'extract_file'

hash: str = None

mode: int = None

mtime: int = None

path: PurePosixPath

rel_path: str = None

subtype: str

class ksconf.app.deploy.**DeployAction_RemoveFile**(*path: 'PurePosixPath'*)

Bases: DeployAction

action: str = 'remove'

path: PurePosixPath

class ksconf.app.deploy.**DeployAction_SetAppName**(*name: 'str'*)

Bases: DeployAction

action: str = 'app'

name: str

class ksconf.app.deploy.**DeployAction_SourceReference**(*archive_path: 'str', hash: 'str'*)

Bases: DeployAction

action: str = 'source'

archive_path: str

hash: str

class ksconf.app.deploy.**DeployApply**(*dest: Path*)

Bases: object

apply_sequence(*deployment_sequence:* DeploySequence)

Apply a pre-calculated deployment sequence to the local file system.

Note that we implicitly trust paths contained within deployment_sequence as all constructors run the check_paths() method on all input manifests. Deployment sequences are created locally and never persisted or transmitted.

resolve_source(*source*, *hash*)

class ksconf.app.deploy.**DeployPlanner**

Bases: object

class ksconf.app.deploy.**DeploySequence**

Bases: object

add(*action: str*, **args*, ***kwargs*)

classmethod from_dict(*data: dict*) → *DeploySequence*

classmethod from_manifest(*manifest:* AppManifest) → *DeploySequence*

Fresh deploy of an app from scratch.

(There should probably be a new op-code for this, eventually instead of listing every single file.)

classmethod from_manifest_transformation(*base:* AppManifest, *target:* AppManifest) → *DeploySequence*

to_dict() → dict

ksconf.app.deploy.**expand_archive_by_manifest**(*archive: Path*, *dest: Path*, *manifest:* AppManifest, *dir_mode=504*)

Expand an tarball to a local file system including only the files referenced by the files within the app manifest.

This function assumes that safety checks on manifest have already been performed, such as eliminating any absolute paths.

ksconf.app.deploy.**get_deploy_action_class**(*action: str*) → *DeployAction*

ksconf.app.facts module

Splunk Application facts:

Easily collect Splunk app name, version, label, and other nuggets from `app.conf`

class ksconf.app.facts.**AppFacts**(*name: str*, *label: str = None*, *id: str = None*, *version: str = None*, *author: str = None*, *description: str = None*, *state: str = None*, *build: int = None*)

Bases: `object`

Basic Splunk application info container. A majority of these facts are extracted from `app.conf`

allows_disable: bool = None

author: str = None

build: int = None

check_for_updates: bool = None

deployer_lookups_push_mode: str = None

deployer_push_mode: str = None

description: str = None

classmethod from_app_dir(*app_path: Path*) → *AppFacts*

Create an AppFacts from a local file system. This expects a standard (non-layered) installed or extracted app folder. Both `default` and `local` are considered.

classmethod from_archive(*archive: Path*)

Returns list of app names, merged app_conf and a dictionary of extra facts that may be useful

classmethod from_conf(*name*, *conf: Dict[str, Dict[str, str]]*) → *AppFacts*

Create AppFacts from an app.conf configuration content.

id: str = None

install_source_checksum: str = None

install_source_local_checksum: str = None

is_configured: bool = None

is_visible: bool = None

label: str = None

name: str

state: str = None

state_change_requires_restart: bool = None

to_dict() → dict

to_tiny_dict(**keep_attrs*) → dict

Return dict representation, discarding the Nones

version: str = None

ksconf.app.manifest module

Splunk App content inventory and signature management

exception ksconf.app.manifest.**AppArchiveContentError**

Bases: Exception

Problem with the contents of an archive

exception ksconf.app.manifest.**AppArchiveError**

Bases: Exception

class ksconf.app.manifest.**AppManifest**(*name: 'str' = None*, *source: 'str' = None*, *hash_algorithm: 'str' = 'sha256'*, *files: 'List[AppManifestFile]' = <factory>*)

Bases: object

check_paths()

Check for dangerous paths in the archive.

files: list[AppManifestFile]

find_local() → Iterable[*AppManifestFile*]

classmethod from_archive(*archive: Path*, *calculate_hash=True*) → *AppManifest*

Create as new AppManifest from a tarball. Set calculate_hash as False when only a file listing is needed.

classmethod from_dict(*data: dict*) → *AppManifest*

classmethod from_filesystem(*path: Path*, *name: str = None*, *follow_symlinks=False*, *calculate_hash=True*) → *AppManifest*

Create as new AppManifest from an existing directory structure. Set calculate_hash as False when only a file listing is needed.

property hash

hash_algorithm: str = 'sha256'

name: str = None

recalculate_hash() → bool

Recalculate hash and indicate if hash has changed.

source: str = None

to_dict()

class ksconf.app.manifest.**AppManifestFile**(*path: 'PurePosixPath'*, *mode: 'int'*, *size: 'int'*, *hash: 'str' = None*)

Bases: object

content_match(*other*)

classmethod from_dict(*data: dict*) → *AppManifestFile*

hash: str = None

mode: int

path: PurePosixPath

size: int

to_dict()

exception ksconf.app.manifest.**AppManifestStorageError**

Bases: Exception

exception ksconf.app.manifest.**AppManifestStorageInvalid**

Bases: AppManifestStorageError

class ksconf.app.manifest.**StoredArchiveManifest**(*archive: Path*, *size: int*, *mtime: float*, *hash: str*)

Bases: object

Stored manifest for a tarball. Typically the manifest file lives in the same directory as the archive. Details around the naming, storage, and clean up of these persistent manifest files are managed by the caller.

archive: Path

classmethod from_dict(*data: dict*) → *StoredArchiveManifest*

classmethod from_file(*archive: Path*, *manifest:* AppManifest) → *StoredArchiveManifest*

Construct instance from a tarball.

classmethod from_json_manifest(*archive: Path*, *stored_file: Path*, ***, *permanent_archive: Path = None*) → *StoredArchiveManifest*

Attempt to load an archive stored manifest from `archive` and `stored_file` paths. If the archive has changed since the manifest was stored, then an exception will be raised indicating the reason for invalidation.

hash: str

property manifest: AppManifest

mtime: float

classmethod read_json_manifest(*manifest_file: Path*) → *StoredArchiveManifest*

size: int

to_dict()

write_json_manifest(*manifest_file: Path*)

ksconf.app.manifest.**create_manifest_from_archive**(*archive_file: Path*, *manifest_file: Path*, *manifest:* AppManifest) → *StoredArchiveManifest*

Create a new stored manifest file based on a given archive.

ksconf.app.manifest.**get_stored_manifest_name**(*archive: Path*) → Path

Calculate the name of the stored manifest file based on `archive`.

ksconf.app.manifest.**load_manifest_for_archive**(*archive: ~pathlib.Path*, *manifest_file: ~pathlib.Path = None*, ***, *read_manifest=True*, *write_manifest=True*, *permanent_archive: ~pathlib.Path = None*, *log_callback=<built-in function print>*) → *AppManifest*

Load manifest for `archive` and create a stored copy of the manifest in `manifest_file`. On subsequent calls the manifest data stored to disk will be reused assuming `manifest_file` is up-to-date.

File modification time and size are used to determine if `archive` has been changed since the `manifest_file` was written.

If no `manifest_file` is provided, the default manifest naming convention will be applied where the `manifest_file` is stored in the same directory as `archive`.

If `permanent_archive` is provided, then we assume it is the persistent name and `archive` is a temporary resource. In this mode, the default `manifest_file` is also based on `permanent_archive` not `archive`.

Module contents

Splunk App helper classes

Note that these representations are for native Splunk apps that use 'default' and 'local' and have not built-in concept of ksconf layers.

ksconf.app.**get_facts_manifest_from_archive**(*archive: Path*, *calculate_hash=True*, *check_paths=True*) → tuple[*AppFacts*, *AppManifest*]

Get both AppFacts and AppManifest from a single archive. If `calculate_hash` is True, then the manifest will contain checksums for all files in the archive. Without this, it's not possible to calculate a hash for the combined manifest.

Use this function to collect both metadata about the app and a full listing of the app's contents.

ksconf.builder package

Submodules

ksconf.builder.cache module

class ksconf.builder.cache.**CachedRun**(*root: Path*)

Bases: `object`

STATE_DISABLED = 'disabled'

STATE_EXISTS = 'exists'

STATE_NEW = 'new'

STATE_TAINT = 'taint'

cache_dir

property cached_inputs

property cached_outputs

config_file

disable()

dump()

property exists

inputs_identical(*inputs:* FileSet) → bool

property is_disabled

property is_expired

property is_new

load()

rename(*dest*)

root

set_cache_info(*type: str*, *data:* FileSet)

set_settings(*cache_settings*)

taint()

class ksconf.builder.cache.**FileSet**(*fingerprint=<function fingerprint_hash>*)

Bases: object

A collection of fingerprinted files.

Currently the fingerprint is only a SHA256 hash.

Two constructors are provided for building an instance from either files that live on the filesystem, via from_filesystem() or from a persisted cached record available from the from_cache(). The filesystem version actively reads all inputs files at object creation time, so this can be costly, especially if repeated.

add_file(*root: Path*, *relative_path: str*)

Add a simple relative path to a file to the FileSet.

add_glob(*root: Path*, *pattern: str*)

Recursively add all files matching glob pattern.

copy_all(*src_dir: Path*, *dest_dir: Path*)

Copy a the given set of files from one location to another.

files

files_meta

classmethod from_cache(*data*)

classmethod from_filesystem(*root: Path*, *files: List[str] = None*) → *FileSet*

Create a new FileSet instance based on a filesystem location. If files is None, then the entire directory is added recursively.

get_fingerprint

ksconf.builder.cache.**fingerprint_hash**(*path: Path*) → dict

ksconf.builder.cache.**fingerprint_stat**(*path: Path*) → dict

ksconf.builder.core module

Cache build requirements:

- Caching mechanism should inspect 'inputs' (collect file hashes) to determine if any content has changed. If input varies, then command should be re-run.
- Command (decorated function) should be generally unaware of all other details of build process, and it should *ONLY* be able to see files listed in "inputs"
- Allow caching to be fully disabled (run in-place with no dir proxy) for CI/CD
- Cache should have allow a timeout parameter

decorator used to implement caching:

- **decorator args:**
  - inputs: list or glob
  - **outputs (do we need this, can we just detect this??)**
    Default to "." (everything)
  - timeout=0 Seconds before cache should be considered stale
  - **name=None If not given, default to the short name of the function.**
    (Cache "slot"), must be filesystem safe]

class ksconf.builder.core.**BuildManager**

Bases: object

Supports an application building process by managing individual build steps

New in version v0.8.0.

cache(*inputs: List[str]*, *outputs: int*, *timeout: int = None*, *name: str = None*, *cache_invalidation: dict | list | str = None*) → None

function decorator for caching build steps Wrapped function must accept BuildStep instance as first parameters

XXX: Clearly document what things are good cache candidates and which are not.

Example:

- No extra argument to the function (at least currently)
- Changes to inputs files are not supported
- Deleting files aren't supported
- Can only operate in a single directory given a limited set of inputs
- Cannot read from the source directory, and agrees not to write to dist (In other words, limit all activities to build_path for deterministic behavior)

disable_cache()

get_build_step(*output=None*) → *BuildStep*

get_cache_info(*name: str*) → *CachedRun*

is_folders_set()

set_folders(*source_path: Path*, *build_path: Path*, *dist_path: Path = None*)

taint_cache()

ksconf.builder.steps module

ksconf.builder.steps: Collection of reusable build steps for reuse in your build script.

ksconf.builder.steps.**clean_build**(*step:* BuildStep) → None

Ensure a clean build folder for consistent build results.

ksconf.builder.steps.**copy_files**(*step:* BuildStep, *patterns: List[str]*, *target: str = None*) → None

Copy source files into the build folder that match given glob patterns

ksconf.builder.steps.**pip_install**(*step:* BuildStep, *requirements_file: str = 'requirements.txt'*, *dest: str = 'lib'*, ***, *python_path: str = None*, *isolated: bool = True*, *dependencies: bool = True*, *handle_dist_info: str = 'remove'*, *remove_console_scripts: bool = True*) → None

Module contents

exception ksconf.builder.**BuildCacheException**

Bases: Exception

exception ksconf.builder.**BuildExternalException**

Bases: Exception

class ksconf.builder.**BuildStep**(*build: ~pathlib.Path*, *source: ~pathlib.Path = None*, *dist: ~pathlib.Path = None*, *output: ~typing.TextIO = <_io.TextIOWrapper name='<stdout>' mode='w' encoding='utf-8'>*)

Bases: object

alternate_path(*path*) → *BuildStep*

Construct a new BuildStep instance with only 'build_path' altered.

build_path

config

dist_path

get_logger(*prefix: str = None*) → Callable

property is_quiet

is_verbose()

run(*executable*, **args*, *cwd=None*)

Execute an OS-level command regarding the build process. The process will run withing the working directory of the build folder.

Parameters

- **executable** (str) – Executable to launch for a build step.
- **args** (str) – Additional argument(s) for the new process.
- **cwd** (str) – Optional kw arg to change the working directory. This defaults to the build folder.

run_ksconf(**args*, *cwd=None*)

Execute 'ksconf' command in the build folder. Currently this runs as a separate process, but in the future is may be optimized to run from within the same python process. This is an implementation detail the caller shouldn't care about.

Parameters

- **args** (str) – Additional argument(s) for the ksconf command.
- **cwd** (str) – Optional kw arg to change the working directory. This defaults to the build folder.

source_path

verbosity

ksconf.builder.**default_cli**(*build_manager:* BuildManager, *build_funct: Callable*, *argparse_parents: List[ArgumentParser] = ()*)

This is the function you stick in the: `if __name__ == '__main__'` section of your code :-)

Pass in a BuildManager instance, and a callback function. The callback function must accept (steps, args). If you have need for custom arguments, you can add them to your own ArgumentParser instance and pass them to the argparse_parents keyword argument, and then handle additional 'args' passed into the callback function.

ksconf.commands namespace

Submodules

ksconf.commands.attr module

SUBCOMMAND: `ksconf attr-get <CONF> --stanza STANZA --attribute ATTR`

```
ksconf attr-get $SPLUNK_HOME/etc/apps/Splunk_TA_aws/default/app.conf --stanza␣
↪launcher --attribute version
```

SUBCOMMAND: `ksconf attr-set <CONF> --stanza STANZA --attribute ATTR --value VALUE`

ksconf attr-set $SPLUNK_HOME/etc/apps/Splunk_TA_aws/local/app.conf –stanza launcher –attribute version –value 9.9.9

echo "9.9.9" > /tmp/new_version ksconf attr-set $SPLUNK_HOME/etc/apps/Splunk_TA_aws/local/app.conf –stanza launcher –attribute version -t file /tmp/new_version

export NEW_VERSION=1.2.3 ksconf attr-set $SPLUNK_HOME/etc/apps/Splunk_TA_aws/local/app.conf –stanza launcher –attribute version -t env NEW_VERSION

class ksconf.commands.attr.**AttrGetCmd**(*name*)

Bases: KsconfCmd

description = 'Get a specific stanza and attribute value from a Splunk .conf file.\n'

format = 'manual'

help = 'Get the value from a specific stanzas and attribute'

maturity = 'beta'

pre_run(*args*)

Optional pre-run hook. Any exceptions or non-0 return code, will prevent run()/post_run() from being called.

register_args(*parser*)

This function in passed the

run(*args*)

For a given conf file, get the 'value' from [stanza] attribute = value

class ksconf.commands.attr.**AttrSetCmd**(*name*)

Bases: KsconfCmd

description = 'Set a specific stanza and attribute value of a Splunk .conf file.\nThe value can be provided as a command line argument, file, or\nenvironment variable\n\nThis command does not support preserving leading or trailing whitespace.\nNormally this is desireable.\n'

format = 'manual'

get_value(*value*, *value_type*)

help = 'Set the value of a specific stanzas and attribute'

maturity = 'beta'

register_args(*parser*)

This function in passed the

run(*args*)

For a given conf file, set [stanza] attribute = value

set_conf_value(*conf_file: Path*, *stanza: str*, *attribute: str*, *value: str*, *create_missing: bool*, *no_overwrite: bool*)

ksconf.commands.check module

SUBCOMMAND: `ksconf check <CONF>`

Usage example: (Nice pre-commit script)

```
find . -name '*.conf' | ksconf check -
```

class ksconf.commands.check.**CheckCmd**(*name*)

Bases: KsconfCmd

description = "\nProvides basic syntax and sanity checking for Splunk's .conf\nfiles. Use Splunk's built-in ``btool check`` for a more robust\nvalidation of attributes and values.\n\nConsider using this utility as part of a pre-commit hook."

help = 'Perform basic syntax and sanity checks on .conf files'

maturity = 'stable'

pre_run(*args*)

Optional pre-run hook. Any exceptions or non-0 return code, will prevent run()/post_run() from being called.

register_args(*parser*)

This function in passed the

run(*args*)

Actual works happens here. Return code should be an EXIT_CODE_* from consts.

ksconf.commands.combine module

SUBCOMMAND: `combine --target=<DIR> <SRC1> [ <SRC-n> ]`

Usage example:

```
cd MY_APP
ksconf combine default.d/* --target=default
```

class ksconf.commands.combine.**CombineCmd**(*name*)

Bases: KsconfCmd

description = "Merge .conf settings from multiple source directories into a combined target\ndirectory. Configuration files can be stored in a ``/etc/*.d`` like directory\nstructure and consolidated back into a single 'default' directory.\n\nThis command supports both one-time operations and recurring merge jobs. For\nexample, this command can be used to combine all users' knowledge objects (stored\nin 'etc/users') after a server migration, or to merge a single user's settings\nafter their account has been renamed. Recurring operations assume some type\nof external scheduler is being used. A best-effort is made to only write to\ntarget files as needed.\n\nThe 'combine' command takes your logical layers of configs (upstream, corporate,\nSplunk admin fixes, and power user knowledge objects, ...) expressed as\nindividual folders and merges them all back into the single ``default`` folder\nthat Splunk reads from. One way to keep the 'default' folder up-to-date is\nusing client-side git hooks.\n\nNo directory layout is mandatory, but taking advantages of the native-support\nfor 'dir.d' layout works well for many uses cases. This idea is borrowed from\nthe Unix System V concept where many services natively read their config files\nfrom ``/etc/*.d`` directories.\n\nVersion notes: dir.d was added in ksconf 0.8. Starting in 1.0 the default will\nswitch to 'dir.d', so if you need the old behavior be sure to update your scripts.\n"

format = 'manual'

help = 'Combine configuration files across multiple source directories into a single\ndestination directory. This allows for an arbitrary number of Splunk\nconfiguration layers to coexist within a single app. Useful in both ongoing\nmerge and one-time ad-hoc use.\n'

maturity = 'beta'

register_args(*parser*)

This function in passed the

run(*args*)

Actual works happens here. Return code should be an EXIT_CODE_* from consts.

exception ksconf.commands.combine.**LayerCombinerExceptionCode**(*msg*, *return_code=None*)

Bases: LayerCombinerException

class ksconf.commands.combine.**RepeatableCombiner**(**args*, *disable_marker: bool = False*, *disable_cleanup: bool = False*, *keep_existing: bool = False*, ***kwargs*)

Bases: LayerCombiner

Re-runable combiner class. Beyond the reusable layer combining functionality, this class enables the use of a marker file for added safety. Removed files will cleanup.

post_combine(*target: Path*)

Handle cleanup of extra files

pre_combine_inventory(*target: Path*, *src_files: list[*LayerFile*]*) → list[*LayerFile*]

Find a set of files that exist in the target folder, but in NO source folder (for cleanup)

prepare_target_dir(*target: Path*)

Handle marker file and ensure that target directory gets created safely.

ksconf.commands.diff module

SUBCOMMAND: `ksconf diff <CONF> <CONF>`

Usage example:

```
ksconf diff default/props.conf default/props.conf
```

class ksconf.commands.diff.**DiffCmd**(*name*)

Bases: KsconfCmd

description = "Compares the content differences of two .conf files\n\nThis command ignores textual differences (like order, spacing, and comments) and\nfocuses strictly on comparing stanzas, keys, and values. Note that spaces within\nany given value, will be compared. Multi-line fields are compared in a more traditional\n'diff' output so that long saved searches and macros can be compared more easily.\n"

format = 'manual'

help = 'Compare settings differences between two .conf files ignoring spacing and sort order'

maturity = 'stable'

register_args(*parser*)

This function in passed the

run(*args*)

Compare two configuration files.

ksconf.commands.filter module

SUBCOMMAND: `ksconf filter <CONF>`

Usage example:

```
ksconf filter default/savedsearches.conf --stanza "My Special Search" -o my-special-
↪search.conf
```

Future things to support:

- SED-like rewriting for stanza name or key values.
- Mini eval/query language for simple data manipulations supporting mixed used of matching modes on a case-by-base basis, custom logic (AND,OR,arbitrary groups), projections, and content rewriting. (Should leverage custom 'combine' mini-language where possible.)

class ksconf.commands.filter.**FilterCmd**(**args*, ***kwargs*)

Bases: KsconfCmd

description = '\nFilter the contents of a conf file in various ways. Stanzas can be included\nor excluded based on a provided filter or based on the presence or value of a key.\n\nWhere possible, this command supports GREP-like arguments to bring a familiar feel.\n'

filter_attrs(*content: dict*) → dict

help = 'A stanza-aware GREP tool for conf files'

maturity = 'alpha'

output(*args*, *matches: dict*, *filename*)

Process output for a single input file.

prep_filters(*args*)

register_args(*parser: ArgumentParser*)

This function in passed the

run(*args*)

Filter configuration files.

ksconf.commands.filter.**is_disabled**(*attributes: dict*) → bool

ksconf.commands.merge module

SUBCOMMAND: `ksconf merge --target=<TARGET_CONF> <CONF> [ <CONF-n> ... ]`

Usage example:

```
ksconf merge --target=master-props.conf /opt/splunk/etc/apps/*TA*/{default,local}/
↪props.conf
```

class ksconf.commands.merge.**MergeCmd**(*name*)

Bases: KsconfCmd

description = 'Merge two or more .conf files into a single combined .conf file.\nThis is similar to the way that Splunk logically combines the ``default`` and ``local``\nfolders at runtime.\n'

help = 'Merge two or more .conf files'

maturity = 'stable'

pre_run(*args*)

Optional pre-run hook. Any exceptions or non-0 return code, will prevent run()/post_run() from being called.

register_args(*parser*)

This function in passed the

run(*args*)

Merge multiple configuration files into one

ksconf.commands.minimize module

SUBCOMMAND: `ksconf minimize --target=<CONF> <CONF> [ <CONF-n> ... ]`

Usage example:

```
ksconf minimize --target=local/inputs.conf default/inputs.conf
```

Example workflow:

1. `cp default/props.conf local/props.conf`
2. `vi local/props.conf` (edit JUST the lines you want to change)
3. `ksconf minimize --target=local/props.conf default/props.conf` (You could take this a step further by appending "$SPLUNK_HOME/system/default/props.conf" and removing any SHOULD_LINEMERGE = true entries (for example)

class ksconf.commands.minimize.**MinimizeCmd**(*name*)

Bases: KsconfCmd

description = "Minimize a conf file by removing any duplicated default settings.\n\nReduce a local conf file to only your intended changes without manually tracking\nwhich entries you've edited. Minimizing local conf files makes your local\ncustomizations easier to read and often results in cleaner upgrades.\n"

help = 'Minimize the target file by removing entries duplicated in the default conf(s)'

maturity = 'beta'

register_args(*parser*)

This function in passed the

run(*args*)

Actual works happens here. Return code should be an EXIT_CODE_* from consts.

ksconf.commands.minimize.**explode_default_stanza**(*conf*, *default_stanza=None*)

Take the GLOBAL stanza, (aka [default]) and apply it's settings underneath ALL other stanzas. This is mostly only useful in minimizing and other comparison operations.

ksconf.commands.package module

SUBCOMMAND: `ksconf package -f <SPL> <DIR>`

Usage example:

```
ksconf package -f myapp.tgz MyApp/
```

Build system example:

```
ksconf package -f release/myapp-{{version}}.tgz \
       --block-local \
       --set-version={{git_tag}} \
       --set-build=${TRAVIS_BUILD_NUMBER:-0}
```

class ksconf.commands.package.**PackageCmd**(*name*)

Bases: KsconfCmd

default_blocklist = ['.git*', '*.py[co]', '__pycache__', '.DS_Store']

description = 'Create a Splunk app or add on tarball (``.spl``) file from an app directory.\n\n``ksconf package`` can do useful things like, exclude unwanted files, combine layers, set the\napplication version and build number, drop or promote the ``local`` directory into ``default``.\n\nNote that some arguments, like the ``FILE`` support special values that can be automatically\nevaluated at runtime. For example the placeholders ``{{version}}`` or ``{{git_tag}}`` can be\nexpanded into the output tarball filename.\n\nIf both layering and templating are in use at the same time, be aware that templates are\nrendered prior to layering operations. This allows, for example, one layer to include a simple\n``indexes.conf`` file and another layer to include an ``indexes.conf.j2`` template.\n'

help = 'Create a Splunk app .spl file from a source directory'

static load_blocklist(*path*)

maturity = 'beta'

pre_run(*args*)

Optional pre-run hook. Any exceptions or non-0 return code, will prevent run()/post_run() from being called.

register_args(*parser: ArgumentParser*)

This function in passed the

run(*args*)

Create a Splunk app/add-on .spl file from a directory

ksconf.commands.promote module

SUBCOMMAND: `ksconf promote <SOURCE> <TARGET>`

Usage example: Promote local props changes (made via the UI) to the 'default' folder

```
ksconf local/props.conf default/props.conf
```

class ksconf.commands.promote.**PromoteCmd**(*name*)

Bases: KsconfCmd

apply_filters(*delta*, *invert_match=False*)

static combine_stanza(*a*, *b*)

description = 'Propagate .conf settings applied in one file to another. Typically this is used\nto move ``local`` changes (made via the UI) into another layer, such as the\n``default`` or a named ``default.d/50-xxxxx``) folder.\n\nPromote has two modes: batch and interactive. In batch mode, all changes are\napplied automatically and the (now empty) source file is removed. In interactive\nmode, the user is prompted to select stanzas to promote. This way local changes\ncan be held without being promoted.\n\nNOTE: Changes are *MOVED* not copied, unless ``--keep`` is used.\n'

format = 'manual'

help = 'Promote .conf settings between layers using either batch or interactive mode.\n\nFrequently this is used to promote conf changes made via the UI (stored in\nthe ``local`` folder) to a version-controlled directory, such as ``default``.\n'

maturity = 'beta'

prep_filters(*args*)

register_args(*parser: ArgumentParser*)

This function in passed the

run(*args*)

Actual works happens here. Return code should be an EXIT_CODE_* from consts.

ksconf.commands.promote.**empty_dict**(*d*)

ksconf.commands.restexport module

SUBCOMMAND: `ksconf rest-export --output=script.sh <CONF>`

Usage example:

```
ksconf rest-export --output=apply_props.sh /opt/splunk/etc/app/Splunk_TA_aws/local/
↪props.conf
```

NOTE:

> If we add support for Windows CURL, then we'll need to also support proper quoting for the '%' character. This can be done with '%^', wonky, I know...

class ksconf.commands.restexport.**CurlCommand**

Bases: `object`

extend_args(*args*)

get_command()

classmethod quote(*s*)

class ksconf.commands.restexport.**Literal**(*value*)

Bases: `object`

class ksconf.commands.restexport.**RestExportCmd**(*name*)

Bases: KsconfCmd

static build_rest_url(*base*, *owner*, *app*, *conf*)

description = "Build an executable script of the stanzas in a configuration file that can be later applied to\na running Splunk instance via the Splunkd REST endpoint.\n\nThis can be helpful when pushing complex props and transforms to an instance where you only have\nUI access and can't directly publish an app.\n\n"

format = 'manual'

help = 'Export .conf settings as a curl script to apply to a Splunk instance later (via REST)'

maturity = 'deprecated'

register_args(*parser: ArgumentParser*)

This function in passed the

run(*args*)

Convert a conf file into a bunch of CURL commands

ksconf.commands.restpublish module

SUBCOMMAND: `ksconf rest-publish <ENDPOINT> <CONF>`

Usage example:

```
ksconf rest-publish MyApp/local/props.conf
```

class ksconf.commands.restpublish.**RestPublishCmd**(**args*, ***kwargs*)

Bases: KsconfCmd

connect_splunkd(*args: Namespace*)

delete_conf(*stanza_name: str*, *stanza_data: Dict[str, Dict[str, str]]*, *config_file*)

description = "Publish stanzas in a .conf file to a running Splunk instance via REST. This requires access to\nthe HTTPS endpoint of Splunk. By default, ksconf will handle both the creation of new stanzas\nand the update of existing stanzas.\n\nThis can be used to push full configuration stanzas where you only have REST access and can't\ndirectly publish an app.\n\nOnly attributes present in the conf file are pushed. While this may seem obvious, this fact can\nhave profound implications in certain situations, like when using this command for continuous\nupdates. This means that it's possible for the source .conf to ultimately differ from what ends\nup on the server's .conf file. One way to avoid this, is to explicitly remove an object using\n``--delete`` mode first, and then insert a new copy of the object. Of course, this means that\nthe object will be unavailable. The other impact is that diffs only compares and shows a subset\nof attribute.\n\nBe aware, that for consistency, the configs/conf-TYPE endpoint is used for this command.\nTherefore, a reload may be required for the server to use the published config settings.\n"

handle_conf_file(*args: Namespace*, *conf_proxy: ConfFileProxy*)

help = 'Publish .conf settings to a live Splunk instance via REST'

static make_boolean(*stanza: Dict[str, Dict[str, str]]*, *attr: str = 'disabled'*)

maturity = 'alpha'

publish_conf(*stanza_name: str*, *stanza_data: Dict[str, Dict[str, str]]*, *config_file*)

register_args(*parser: ArgumentParser*)

This function in passed the

run(*args: Namespace*)

Actual works happens here. Return code should be an EXIT_CODE_* from consts.

ksconf.commands.snapshot module

SUBCOMMAND: `ksconf snapshot --output=FILE.json <PATH> [ ... <PATH-n> ]`

Usage example:

```
ksconf snapshot --output=daily.json /opt/splunk/etc/app/
```

class ksconf.commands.snapshot.**ConfSnapshot**(*config*)

Bases: object

schema_version = 0.2

snapshot_dir(*path*)

snapshot_file_conf(*path*)

write_snapshot(*stream*, ***kwargs*)

class ksconf.commands.snapshot.**ConfSnapshotConfig**

Bases: object

max_file_size = 10485760

class ksconf.commands.snapshot.**SnapshotCmd**(*name*)

Bases: KsconfCmd

description = 'Build a static snapshot of various configuration files stored within a structured json export\nformat. If the .conf files being captured are within a standard Splunk directory structure,\nthen certain metadata and namespace information is assumed based on typical path locations.\nIndividual apps or conf files can be collected as well, but less metadata may be extracted.\n'

help = 'Snapshot .conf file directories into a JSON dump format'

register_args(*parser*)

This function in passed the

run(*args*)

Snapshot multiple configuration files into a single json snapshot.

ksconf.commands.sort module

SUBCOMMAND: `ksconf sort <CONF>`

Usage example: To recursively sort all files (in-place):

```
find . -name '*.conf' | xargs ksconf sort -i
```

class ksconf.commands.sort.**SortCmd**(*name*)

Bases: KsconfCmd

description = 'Sort a Splunk .conf file. Sort has two modes: (1) by default, the sorted\nconfig file will be echoed to the screen. (2) the config files are updated\nin-place when the ``-i`` option is used.\n\nManually managed conf files can be protected against changes by adding a comment containing the\nstring ``KSCONF-NO-SORT`` to the top of any .conf file.\n'

format = 'manual'

help = 'Sort a Splunk .conf file creating a normalized format appropriate for version control'

maturity = 'stable'

pre_run(*args*)

Optional pre-run hook. Any exceptions or non-0 return code, will prevent run()/post_run() from being called.

register_args(*parser*)

This function in passed the

run(*args*)

Sort one or more configuration file.

ksconf.commands.unarchive module

SUBCOMMAND: `ksconf unarchive <tarball>`

Usage example:

```
ksconf unarchive splunk-add-on-for-amazon-web-services_111.tgz
```

class ksconf.commands.unarchive.**UnarchiveCmd**(*name*)

Bases: KsconfCmd

description = "\nInstall or overwrite an existing app in a git-friendly way.\nIf the app already exists, steps will be taken to upgrade it safely.\n\nThe ``default`` folder can be redirected to another path (i.e., ``default.d/10-upstream`` or\nother desirable path if you're using the ``ksconf combine`` tool to manage extra layers).\n"

format = 'manual'

help = 'Install or upgrade an existing app in a git-friendly and safe way'

maturity = 'beta'

register_args(*parser*)

This function in passed the

run(*args*)

Install / upgrade a Splunk app from an archive file

ksconf.commands.xmlformat module

SUBCOMMAND: `ksconf xml-format <XML>`

Usage example: (Nice pre-commit script)

```
find default/data/ui -name '*.xml' | ksconf xml-format -
```

class ksconf.commands.xmlformat.**XmlFormatCmd**(*name*)

Bases: KsconfCmd

description = "\nNormalize and apply consistent XML indentation and CDATA usage for XML dashboards and\nnavigation files.\n\nTechnically this could be used on *any* XML file, but certain element names specific to Splunk's\nsimple XML dashboards are handled specially, and therefore could result in unusable results.\n\nThe expected indentation level is guessed based on the first element indentation, but can be\nexplicitly set if not detectable.\n"

help = 'Normalize XML view and nav files'

maturity = 'alpha'

pre_commit_repo_migration_warning(*args*)

Issue migration warning if (1) running hooks from the old repo (missing arg), and (2) parent process is from pre-commit (env var).

Another workaround is to use:

```
- repo: https://github.com/Kintyre/ksconf
rev: v0.11.8
hooks:
    - id: ksconf-check
    - id: ksconf-sort
    exclude: logging\.conf
    - id: ksconf-xml-format
      args: --disable-pre-commit-migration-check
additional_dependencies: [lxml]
```

But honestly, isn't it just easy to add `-pre-commit` to the repo?

Remove this after Dec 2024 or v0.13.0

register_args(*parser*)

This function in passed the

run(*args*)

Actual works happens here. Return code should be an EXIT_CODE_* from consts.

ksconf.conf package

Submodules

ksconf.conf.delta module

class ksconf.conf.delta.**DiffGlobal**(*type*)

Bases: NamedTuple

type: DiffLevel

Alias for field number 0

class ksconf.conf.delta.**DiffHeader**(*name*, *mtime=None*)

Bases: object

detect_mtime()

mtime: float = None

name: str

class ksconf.conf.delta.**DiffLevel**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

GLOBAL = 'global'

KEY = 'key'

STANZA = 'stanza'

class ksconf.conf.delta.**DiffOp**(*tag*, *location*, *a*, *b*)

Bases: NamedTuple

a: Dict[str, Dict[str, str]] | Dict[str, str] | str | None

Alias for field number 2

b: Dict[str, Dict[str, str]] | Dict[str, str] | str | None

Alias for field number 3

location: DiffGlobal | DiffStanza | DiffStzKey

Alias for field number 1

tag: DiffVerb

Alias for field number 0

class ksconf.conf.delta.**DiffStanza**(*type*, *stanza*)

Bases: NamedTuple

stanza: str

Alias for field number 1

type: DiffLevel

Alias for field number 0

class ksconf.conf.delta.**DiffStzKey**(*type*, *stanza*, *key*)

Bases: NamedTuple

key: str

Alias for field number 2

stanza: str

Alias for field number 1

type: DiffLevel

Alias for field number 0

class ksconf.conf.delta.**DiffVerb**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

DELETE = 'delete'

EQUAL = 'equal'

INSERT = 'insert'

REPLACE = 'replace'

ksconf.conf.delta.**compare_cfgs**(*a: Dict[str, Dict[str, str]]*, *b: Dict[str, Dict[str, str]]*, *replace_level:* DiffLevel = *DiffLevel.GLOBAL*) → List[*DiffOp*]

Calculate a set of deltas which describes how to transform a into b.

Parameters

- **a** (dict) – the first/original configuration entity
- **b** (dict) – the second/target configuration entity
- **replace_level** (str: global, stanza, or key) – The highest level 'replace' event that can be returned. Acceptable values are global, stanza, and key. These examples may help:
  - Using 'global' with identical inputs will report a single global-level equal op.
  - Using 'stanza' with identical inputs will return all stanzas as equal.
  - Using 'key' will ensure that two stanzas with no common keys will be reported in terms of key changes. Whereas 'global' or 'stanza' would result in a single giant replace op.

Returns
a sequence of differences in tuples

Return type
[*DiffOp*]

Note: The `DiffOp` output idea was borrowed from `SequenceMatcher` class in the `difflib` in the standard Python module.

This function returns a sequence of 5 element tuples describing the transformation based on the detail level specified in `replace_level`.

Each `DiffOp` (named tuple) takes the form:

(tag, location, a, b)

tag:

Value	Meaning
'replace'	same element in both, but different values.
'delete'	remove value b
'insert'	insert value a
'equal'	same values in both

location is a namedtuple that can take the following forms:

Tuple form	Type	Description
("global")	DiffGlobal	Global file level context (e.g., both files are the same)
("stanza", stanza)	DiffStanza	Stanzas are the same, or completely different (no shared keys)
("key", stanza, key)	DiffStzKey	Key level change

Changed in version v0.8.8: The `preserve_empty` argument was originally introduced to preserve backwards compatibility, but it ended up introducing new bugs. Additionally, no use cases were found where better to automatically discarding empty stanzas.

Changed in version v0.8.8: The `allow_level0` argument was replaced with `replace_level`. Instead of using `allow_level0=False` use `replace_level="stanza"`. At the same time a new feature was added to support `replace_level="key"`. The default behavior remains the same.

ksconf.conf.delta.**compare_stanzas**(*a: Dict[str, str]*, *b: Dict[str, str]*, *stanza_name: str*, *replace_level:* DiffLevel = *DiffLevel.GLOBAL*) → List[*DiffOp*]

Parameters

replace_level (bool) – If a and b have no common keys, is a single stanza-level 'replace' is issue unless `replace_level="key"`

ksconf.conf.delta.**diff_obj_json_format**(*o*)

ksconf.conf.delta.**is_equal**(*delta: List[DiffOp]*) → bool

Is the delta output show that the compared objects are identical

ksconf.conf.delta.**reduce_stanza**(*stanza: Dict[str, str]*, *keep_attrs: Sequence*) → dict

Pre-process a stanzas so that only a common set of keys will be compared.

Parameters

- **stanza** (dict) – Stanzas containing attributes and values
- **keep_attrs** ((list, set, tuple, dict)) – Listing of attributes to preserve

Returns

a reduced copy of stanza.

ksconf.conf.delta.**show_diff**(*stream: TextIO*, *diffs: List[DiffOp]*, *headers=None*) → int

ksconf.conf.delta.**show_text_diff**(*stream: TextIO*, *a: PathLike*, *b: PathLike*)

ksconf.conf.delta.**summarize_cfg_diffs**(*delta: List[DiffOp]*, *stream: TextIO*)

Summarize a delta into a human-readable format. The input `delta` is in the format produced by the compare_cfgs() function.

ksconf.conf.delta.**write_diff_as_json**(*delta: List[DiffOp]*, *stream*, ***dump_args*)

ksconf.conf.merge module

ksconf.conf.merge.**merge_app_local**(*app_folder: str*, *cleanup: bool = True*) → None

Find everything in local, if it has a corresponding file in default, merge. This function assumes standard Splunk app path names.

ksconf.conf.merge.**merge_conf_dicts**(**dicts: Dict[str, Dict[str, str]]*) → Dict[str, Dict[str, str]]

ksconf.conf.merge.**merge_conf_files**(*dest: ConfFileProxy*, *configs: List[ConfFileProxy]*, *dry_run: bool = False*, *banner_comment: str = None*) → *SmartEnum*

ksconf.conf.merge.**merge_update_any_file**(*dest: str*, *sources: List[str]*, *remove_source: bool = False*) → None

ksconf.conf.merge.**merge_update_conf_file**(*dest: str*, *sources: List[str]*, *remove_source: bool = False*) → None

Dest is treated as both the output, and the highest priority source.

ksconf.conf.meta module

Incomplete documentation available here:

https://docs.splunk.com/Documentation/Splunk/latest/Admin/Defaultmetaconf

Specifically, attribute-level ACls aren't discussed nor is the magic "import" directive.

LEVELS:

0 - global (or 1 stanza="default") 1 - conf 2 - stanzas 3 - attribute

class ksconf.conf.meta.**MetaData**

Bases: object

static expand_layers(*layers*)

Parameters
layers (list(dict)) – layer of stanzas, starting with the global ending with conf/stanza/attr

Returns
Expanded layer

Return type
dict

feed_conf(*conf*)

feed_file(*stream*)

get(**names*)

get_layer(**names*)

iter_raw()

RAW

classmethod parse_meta(*stanza*)

Split out the values of 'access' (maybe more someday) :param stanza: content of a meta stanza :return: extended meta data :rtype: dict

regex_access = '(?:^|\\s*,\\s*)(?P<action>read|write)\\s*:\\s*\\[\\s*(?P<roles>[^\\]]+?)\\s*\\]'

walk()

write_stream(*stream: TextIO*, *sort=True*)

class ksconf.conf.meta.**MetaLayer**(*name*)

Bases: object

property data

items(*prefix=None*)

Helpful when rebuilding the input file.

resolve(*name*)

update(**args*, ***kwargs*)

walk(*_prefix=()*)

ksconf.conf.parser module

Parse and write Splunk's .conf files

According to this doc:

https://docs.splunk.com/Documentation/Splunk/7.2.3/Admin/Howtoeditaconfigurationfile

1. Comments must start at the beginning of a line (#)
2. Comments may not be after a stanza name or on an attribute's value
3. Supporting encoding is UTF-8 (and therefore ASCII too)

exception ksconf.conf.parser.**ConfParserException**

Bases: Exception

class ksconf.conf.parser.**DuplicateEnum**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

EXCEPTION = 'exception'

MERGE = 'merge'

OVERWRITE = 'overwrite'

exception ksconf.conf.parser.**DuplicateKeyException**

Bases: ConfParserException

exception ksconf.conf.parser.**DuplicateStanzaException**

Bases: ConfParserException

class ksconf.conf.parser.**Token**

Bases: object

Immutable token object. deepcopy returns the same object

ksconf.conf.parser.**conf_attr_boolean**(*value: str | bool | int*) → bool

ksconf.conf.parser.**cont_handler**(*iterable: Iterable[str]*, *continue_re: Pattern = re.compile('^(.*)\\\\$')*, *breaker: str = '\n'*) → Generator[str, None, None]

Look for trailing backslashes ("\") which indicate a value for an attribute is split across multiple lines. This function will group such lines together, and pass all other lines through as-is. Note that the continuation character must be the very last character on the line, trailing whitespace is not allowed.

Parameters

- **iterable** (iter) – lines from a configuration file
- **continue_re** – regular expression to detect the continuation character
- **breaker** – joining string when combining continued lines into a single string. Default '\n'

Returns
lines of text

Return type
str

ksconf.conf.parser.**detect_by_bom**(*path: Path | str*) → str

ksconf.conf.parser.**inject_section_comments**(*section: Dict[str, str]*, *prepend: str = None*, *append: str = None*)

Extract existing comments from section dict (in order; and remove them) Add in any prepend/append comments (if that comment isn't already present) Re-inject comments back into the section dict with fresh numbering

ksconf.conf.parser.**parse_conf**(*stream: Path | str | TextIO | Iterable[str]*, *profile: Dict = {'dup_key': DuplicateEnum.OVERWRITE, 'dup_stanza': DuplicateEnum.EXCEPTION, 'keep_comments': True, 'strict': True}*, *encoding: str = None*) → Dict[str, Dict[str, str]]

Parse a .conf file. This is a wrapper around `parse_conf_stream()` that allows filenames or stream to be passed in.

Parameters

- **stream** (str, file) – the path to a configuration file or open file-like object to be parsed
- **profile** – parsing configuration settings
- **encoding** – Defaults to the system default, (Often "utf-8")

Returns
a mapping of the stanza and attributes. The resulting output is accessible as [stanza][attribute] -> value

Return type
dict

ksconf.conf.parser.**parse_conf_stream**(*stream: TextIO | Iterable[str]*, *keys_lower: bool = False*, *handle_conts: bool = True*, *keep_comments: bool = False*, *dup_stanza:* DuplicateEnum *= DuplicateEnum.EXCEPTION*, *dup_key:* DuplicateEnum *= DuplicateEnum.OVERWRITE*, *strict: bool = False*) → Dict[str, Dict[str, str]]

Low-level conf parsing functionality.

Most often, either `parse_conf()` or `parse_string()` are better options.

ksconf.conf.parser.**parse_string**(*s: str*, *name: str = None*, *profile: Dict = {'dup_key': DuplicateEnum.OVERWRITE, 'dup_stanza': DuplicateEnum.EXCEPTION, 'keep_comments': True, 'strict': True}*) → Dict[str, Dict[str, str]]

Parse a .conf file that's already in memory, as a string.

ksconf.conf.parser.**section_reader**(*stream: Iterable[str]*, *section_re: Pattern = re.compile('^[\\s\\t]*\\[(.*)\\]\\s*$')*) → Generator[Tuple[str, List[str]], None, None]

This generator break a configuration file stream into sections. Each section contains a name and a list of text lines held within that section.

Sections that have no entries must be preserved. Any lines before the first section are send back with the section name of None.

Parameters

- **stream** (file) – configuration file input stream
- **section_re** – regular expression for detecting stanza headers

Returns
sections in the form of *(section_name, lines_of_text)*

Return type
tuple

ksconf.conf.parser.**smart_write_conf**(*filename: Path | str*, *conf: Dict[str, Dict[str, str]]*, *stanza_delim: str = '\n'*, *sort: bool = True*, *temp_suffix: str = '.tmp'*, *mtime: float = None*) → *SmartEnum*

Write conf data to a specific file, but only when necessary. This function is essentially the same as `write_conf()`, except that it avoids updating the file if it already exists and has the desired content.

ksconf.conf.parser.**splitup_kvpairs**(*lines: Iterable[str]*, *comments_re: Pattern = re.compile('^\\s*[#;]')*, *keep_comments: bool = False*, *strict: bool = False*) → Generator[Tuple[str, str], None, None]

Break up 'attribute=value' entries in a configuration file.

Parameters

- **lines** (iter) – the body of a stanza containing associated attributes and values
- **comments_re** – Regular expression used to detect comments.
- **keep_comments** (bool, optional) – Should comments be preserved in the output. Defaults to *False*.
- **strict** (bool, optional) – Should unknown content in the stanza stop processing. Defaults to *False* allowing "junk" to be silently ignored for a best-effort parse.

Returns
iterable of (attribute,value) tuples

class ksconf.conf.parser.**update_conf**(*conf_path: Path | str*, *profile: Dict = {'dup_key': DuplicateEnum.OVERWRITE, 'dup_stanza': DuplicateEnum.EXCEPTION, 'keep_comments': True, 'strict': True}*, *encoding: str = None*, *make_missing: bool = False*)

Bases: object

Context manager that allows for simple in-place updates to conf files. This provides a simple dict-like interface for easy updates.

Usage example:

```
with update_conf("app.conf") as conf:
    conf["launcher"]["version"] = "1.0.2"
    conf["install"]["build"] = 33
```

Parameters

- **conf_path** (str) – Path to .conf file to be edited.
- **profile** (dict) – Parsing settings and strictness profile.
- **encoding** (str) – encoding to use for file operations.
- **make_missing** (bool) – When true, a new blank configuration file will be created if conf_path is missing, otherwise an exception will be raised.

cancel()
Indicate that no updates were made and all processing is complete. An error will occur if additional read/writes are attempted.

keys() → List[str]

update(**args*, ***kwargs*)

ksconf.conf.parser.**write_conf**(*stream: Path | str | TextIO*, *conf: Dict[str, Dict[str, str]]*, *stanza_delim: str = '\n'*, *sort: bool = True*, *temp_suffix: str = '.tmp'*, *mtime: float = None*)

ksconf.conf.parser.**write_conf_stream**(*stream: TextIO*, *conf: Dict[str, Dict[str, str]]*, *stanza_delim: str = '\n'*, *sort: bool = True*)

Module contents

ksconf.util package

Submodules

ksconf.util.compare module

ksconf.util.compare.**cmp_sets**(*a*, *b*)

Result tuples in format (a-only, common, b-only)

ksconf.util.compare.**file_compare**(*fn1*, *fn2*)

ksconf.util.compare.**fileobj_compare**(*f1*, *f2*)

ksconf.util.completers module

ksconf.util.completers.**DirectoriesCompleter**(**args*, ***kwargs*)

ksconf.util.completers.**FilesCompleter**(**args*, ***kwargs*)

ksconf.util.completers.**autocomplete**(**args*, ***kwargs*)

ksconf.util.file module

class ksconf.util.file.**ReluctantWriter**(*path*, **args*, ***kwargs*)

Bases: `object`

Context manager to intelligently handle updates to an existing file. New content is written to a temp file, and then compared to the current file's content. The file file will be overwritten only if the contents changed.

ksconf.util.file.**atomic_open**(*name: Path*, *temp_name: str | Path | Callable | None*, *mode='w'*, ***open_kwargs*) → IO

Context manager to atomically write to a file stream. Like the open() context manager, a file handle returned when the context is entered. Upon successful completion, the temporary file is renamed into place; thus providing an atomic update operation.

See `atomic_writer()` for behaviors regarding the `temp_name` parameter option.

This function can be used nearly any place that `with open(myfile, mode="w") as stream`

ksconf.util.file.**atomic_writer**(*dest: Path*, *temp_name: str | Path | Callable | None*) → str

Context manager to atomically update a destination. When entering the context, a temporary file name is returned. When the context is successfully exited, the temporary file is renamed into place. Either way, the temporary file is removed.

The name of the temporary file can be controlled via `temp_name`. If a `str` is provided, it will be used as a suffix. If a Path is provided, that will be used as the literal temporary file name. If a callable is given, the `dest` path will be passed into the callable to determine the temporary file. Alternatively, the entire _atomic_ nature of this function can be disabled by passing temp_name=None.

ksconf.util.file.**dir_exists**(*directory*)

Ensure that the directory exists

ksconf.util.file.**expand_glob_list**(*iterable*, *do_sort=False*)

ksconf.util.file.**file_fingerprint**(*path*, *compare_to=None*)

ksconf.util.file.**file_hash**(*path*, *algorithm='sha256'*)

ksconf.util.file.**relwalk**(*top*, *topdown=True*, *onerror=None*, *followlinks=False*)

Relative path walker Like os.walk() except that it doesn't include the "top" prefix in the resulting 'dirpath'.

ksconf.util.file.**secure_delete**(*path: Path*, *passes=3*)

A simple file shred technique. If there's demand, this could be expanded. But for now, 'secure' means just slightly more secure that unlink().

Adapted from from Ansible's _shred_file_custom()

ksconf.util.file.**smart_copy**(*src*, *dest*)

Copy (overwrite) file only if the contents have changed.

ksconf.util.file.**splglob_simple**(*pattern*)

Return a splglob that either matches a full path or match a simple file

ksconf.util.file.**splglob_to_regex**(*pattern*, *re_flags=None*)

ksconf.util.rest module

ksconf.util.rest.**build_rest_namespace**(*base*, *owner=None*, *app=None*)

ksconf.util.rest.**build_rest_url**(*base*, *service*, *owner=None*, *app=None*)

ksconf.util.terminal module

class ksconf.util.terminal.**TermColor**(*stream*)

Bases: object

Simple color setting helper class that's a context manager wrapper around a stream. This ensure that the color is always reset at the end of a session.

color(**codes*)

reset()

write(*content*)

Module contents

ksconf.util.**debug_traceback**()

If the 'KSCONF_DEBUG' environmental variable is set, then show a stack trace.

ksconf.util.**decorator_with_opt_kwargs**(*decorator: Callable*) → Callable

Make a decorator that can work with or without args. Heavily borrowed from: https://gist.github.com/ramonrosa/402af55633e9b6c273882ac074760426 Thanks to GitHub user ramonrosa

ksconf.vc package

Submodules

ksconf.vc.git module

class ksconf.vc.git.**GitCmdOutput**(*cmd*, *returncode*, *stdout*, *stderr*, *lines*)

Bases: tuple

cmd

Alias for field number 0

lines

Alias for field number 4

returncode

Alias for field number 1

stderr

Alias for field number 3

stdout

Alias for field number 2

exception ksconf.vc.git.**GitNotAvailable**

Bases: Exception

ksconf.vc.git.**git_cmd**(*args*, *shell=False*, *cwd=None*, *capture_std=True*, *encoding='utf-8'*)

ksconf.vc.git.**git_cmd_iterable**(*args*, *iterable*, *cwd=None*, *cmd_len=1024*)

ksconf.vc.git.**git_is_clean**(*path=None*, *check_untracked=True*, *check_ignored=False*)

ksconf.vc.git.**git_is_working_tree**(*path=None*)

ksconf.vc.git.**git_ls_files**(*path*, **modifiers*)

ksconf.vc.git.**git_status_summary**(*path*)

ksconf.vc.git.**git_status_ui**(*path*, **args*)

ksconf.vc.git.**git_version**()

Module contents

Submodules

ksconf.archive module

class ksconf.archive.**GenArchFile**(*path*, *mode*, *size*, *payload*)

Bases: NamedTuple

mode: int

Alias for field number 1

path: str

Alias for field number 0

payload: bytes | None

Alias for field number 3

size: int

Alias for field number 2

ksconf.archive.**extract_archive**(*archive_name*, *extract_filter: callable = None*) → Iterable[*GenArchFile*]

ksconf.archive.**gaf_filter_name_like**(*pattern*)

ksconf.archive.**gen_arch_file_remapper**(*iterable: Iterable[*GenArchFile*]*, *mapping: Sequence[Tuple[str, str]]*) → Iterable[*GenArchFile*]

ksconf.archive.**sanity_checker**(*iterable: Iterable[*GenArchFile*]*) → Iterable[*GenArchFile*]

ksconf.cli module

KSCONF - Ksconf Splunk CONFig tool

Optionally supports argcomplete for commandline argument (tab) completion.

Install & register with:

pip install argcomplete activate-global-python-argcomplete (in ~/.bashrc)

ksconf.cli.**build_cli_parser**(*do_formatter=False*)

ksconf.cli.**check_py**()

ksconf.cli.**check_py_sane**()

Run a simple python environment sanity check. Here's the scenario, if Splunk's python is called but not all the correct environment variables have been set, then ksconf can fail in unclear ways.

ksconf.cli.**cli**(*argv=None*, *_unittest=False*)

ksconf.cli.**handle_cmd_failed**(*subparser*, *ep*)

Build a bogus subparser for a cmd that can't be loaded, with the only purpose of providing a more consistent user experience.

ksconf.combine module

class ksconf.combine.**LayerCombiner**(*follow_symlink: bool = False*, *banner: str = ''*, *dry_run: bool = False*, *quiet: bool = False*)

Bases: `object`

Class to recursively combine layers (directories) into a single rendered output target directory. This is heavily used by the `ksconf combine` command as well as by the `package` command.

Typical class use case:

::

lc = LayerCombiner()

 # Setup source, either

(1) lc.set_source_dirs() OR

(2) lc.set_layer_root()

Call hierarch:

```
lc.combine()                    Entry point
    -> prepare()                Directory, layer prep
        -> prepare_target_dir() Make dir; subclass handles marker here (combine␣
↪CLI)
    -> pre_combine_inventory()  Hook for pre-processing (or alerting) the set␣
```

(continues on next page)

(continued from previous page)

```
↪of files to combine
   -> combine_files()          Main worker function
   -> post_combine()           Optional, cleanup leftover files
```

add_layer_filter(*action*, *pattern*)

combine(*target: Path*, ***, *hook_label=''*)

Combine layers into `target` directory. Any `hook_label` given will be passed to the plugin system via the `usage` field.

combine_files(*target: Path*, *src_files: list[*LayerFile*]*)

conf_file_re = re.compile('([a-z_-]+\\.conf|(default|local)\\.meta)$')

debug(*message*)

filetype_handlers: list[tuple[Callable, Callable]] = [(<function LayerCombiner.register_handler.<locals>.match_f>, <function handle_merge_conf_files>), (<function LayerCombiner.register_handler.<locals>.match_f>, <function handle_spec_concatenate>)]

log(*message*)

post_combine(*target*)

Hook point for post-processing after all copy/merge operations have been completed.

pre_combine_inventory(*target: Path*, *src_files: list[*LayerFile*]*) → list[*LayerFile*]

Hook point for pre-processing before any files are copied/merged

prepare(*target: Path*)

Start the combine process. This includes directory checking, applying layer filtering, and marker file handling.

prepare_target_dir(*target: Path*)

Hook to ensure destination directory is ready for use. This can be overridden to adder marker file handling for use cases that need it (e.g., the 'combine' command)

classmethod register_handler(*regex_match*)

Decorator that registers a new file type handler. The handler is used if a file name matches a regex. Regex 'search' mode is used.

set_layer_root(*root:* Layer)

set_source_dirs(*sources: list[Path]*)

spec_file_re = re.compile('\\.conf\\.spec$')

exception ksconf.combine.**LayerCombinerException**

Bases: `Exception`

ksconf.combine.**handle_merge_conf_files**(*combiner: LayerCombiner*, *dest_path: Path*, *sources: list[LayerFile]*, *dry_run*)

Handle merging two or more `.conf` files.

ksconf.combine.**handle_spec_concatenate**(*combiner: LayerCombiner*, *dest_path: Path*, *sources: list[LayerFile]*, *dry_run*)

Concatenate multiple `.spec` files. Likely a `README.d` situation.

ksconf.combine.**register_handler**(*regex_match*)

Decorator that registers a new file type handler. The handler is used if a file name matches a regex. Regex 'search' mode is used.

ksconf.command module

ksconf.command:

Helpers functions and classes in support of the actual commands that live under `ksconf.commands.*`.

Note that `ksconf.commands` is a namespace package, which can be contributed to by multiple python packages (technically called "distributions"). Because of this, there can be no __init__.py, which is where this content logically belongs.

class ksconf.command.**ConfFileType**(*mode='r'*, *action='open'*, *parse_profile: Dict = None*, *accept_dir: bool = False*)

Bases: `object`

Factory for creating conf file object types; returns a lazy-loader ConfFile proxy class

Started from FileType() and then changed everything. With our use case, it's often necessary to delay writing, or read before writing to a conf file (depending on whether or not –dry-run mode is enabled, for example.)

Instances of FileType are typically passed as type= arguments to the ArgumentParser add_argument() method.

Parameters

- **mode** (`str`) – How the file is to be opened. Accepts "r", "w", and "r+".
- **action** (`str`) – Determine how much work should be handled during argument parsing vs handed off to the caller. Supports 'none', 'open', 'load'. Full descriptions below.
- **parse_profile** – parsing configuration settings passed along to the parser
- **accept_dir** (`bool`) – Should the CLI accept a directory of config files instead of an individual file. Defaults to *False*.

Values for action

Action	Description
`none`	No preparation or testing is done on the filename.
`open`	Ensure the file exists and can be opened.
`load`	Ensure the file can be opened and parsed successfully.

Once invoked, instances of this class will return a `ConfFileProxy` object, or a `ConfDirProxy` object if a directory is passed in via the CLI.

class ksconf.command.**KsconfCmd**(*name*)

Bases: `object`

Ksconf command specification base class.

add_parser(*subparser*)

description = None

exit(*exit_code*)

Allow overriding for unittesting or other high-level functionality, like an interactive interface.

format = 'default'

help = None

launch(*args*)

Handle flow control between pre_run() / run() / post_run()

maturity = 'alpha'

parse_conf(*path: str*, *mode: str = 'r'*, *profile: Dict = None*, *raw_exec: bool = False*) → ConfFileProxy

parse_extra_vars(*vars: str*, *arg_name='argument'*) → dict

Argument can be either a string, or a @file

post_run(*args*, *exec_info=None*)

Optional custom clean up method. Always called if run() was. The presence of exc_info indicates failure.

pre_run(*args*)

Optional pre-run hook. Any exceptions or non-0 return code, will prevent run()/post_run() from being called.

register_args(*parser: ArgumentParser*)

This function in passed the

run(*args*)

Actual works happens here. Return code should be an EXIT_CODE_* from consts.

version_extra = None

ksconf.command.**add_splunkd_access_args**(*parser: ArgumentParser*) → ArgumentParser

ksconf.command.**add_splunkd_namespace**(*parser: ArgumentParser*) → ArgumentParser

ksconf.command.**dedent**(*text*)

Remove any common leading whitespace from every line in *text*.

This can be used to make triple-quoted strings line up with the left edge of the display, while still presenting them in the source code in indented form.

Note that tabs and spaces are both treated as whitespace, but they are not equal: the lines ” hello” and “thello” are considered to have no common leading whitespace.

Entirely blank lines are normalized to a newline character.

ksconf.command.**get_all_ksconf_cmds**(*on_error='warn'*)

ksconf.command.**get_entrypoints**(*group*, *name=None*)

ksconf.compat module

Silly simple Python version compatibility items

ksconf.compat.**Dict**

alias of dict

ksconf.compat.**List**

alias of list

ksconf.compat.**Set**

alias of set

ksconf.compat.**Tuple**

alias of tuple

ksconf.compat.**cache**(*user_function*, */*)

Simple lightweight unbounded cache. Sometimes called “memoize”.

ksconf.consts module

class ksconf.consts.**SmartEnum**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

CREATE = 'created'

NOCHANGE = 'unchanged'

UPDATE = 'updated'

ksconf.consts.**is_debug**()

ksconf.filter module

class ksconf.filter.**FilteredList**(*flags=0*, *default=True*)

Bases: object

IGNORECASE = 1

INVERT = 2

VERBOSE = 4

feed(*item*, *filter=None*)

feedall(*iterable*, *filter=None*)

property has_rules

match(*item*)

match_path(*path*)

match_stanza(*stanza*)

Same as match(), but handle GLOBAL_STANZA gracefully.

reset_counters()

class ksconf.filter.**FilteredListRegex**(*flags=0*, *default=True*)

Bases: FilteredList

Regular Expression support

calc_regex_flags()

reset_counters()

class ksconf.filter.**FilteredListSplunkGlob**(*flags=0*, *default=True*)

Bases: FilteredListRegex

Classic wildcard support ('*' and ?') plus '...' or '**' for multiple-path components with some (non-advertised) pass-through regex behavior

class ksconf.filter.**FilteredListString**(*flags=0*, *default=True*)

Bases: FilteredList

Handle simple string comparisons

reset_counters()

class ksconf.filter.**FilteredListWildcard**(*flags=0*, *default=True*)

Bases: FilteredListRegex

Wildcard support (handling '*' and ?') Technically fnmatch also supports [] and [!] character ranges, but we don't advertise that

ksconf.filter.**create_filtered_list**(*match_mode: str*, *flags=0*, *default=True*) → *FilteredList*

ksconf.hook module

exception ksconf.hook.**BadPluginWarning**

Bases: UserWarning

Issue with one or more plugins

ksconf.hook.**get_plugin_manager**() → _plugin_manager

Return the shared pluggy PluginManager (singleton) instance.

This is for backwards compatibility. This was only added in v0.11.6; and replaced immediately after.

ksconf.hookspec module

This module contains all the plugin definitions (or hook "specifications") for various customization or integration points with ksconf. Not all of these have been fully tested so please let us know if something is not working as expected, or if additional arguments are needed.

See ksconf plugins on pypi for a list of currently available plugins.

class ksconf.hookspec.**KsconfHookSpecs**(**args*, ***kwargs*)

Bases: Protocol

Ksconf plugin specifications for all known supported functions.

Grouping these functions together in a single class allows for type support it supports typing. This adds a level of validation to the code base where a hook is invoked via `plugin_manger.hook.<hook_name>()`.

If you are implementing one of these hooks, please note that you can simple make top-level function, no need to implement a class.

static ksconf_cli_init()

Simple hook that is run before CLI initialization. This can be use to modify the runtime environment.

This can be used to register additional handlers, such as:

- `ksconf.combine.register_handler()` - Add a combination file handler. File types are limited to pattern matching.
- `ksconf.layer.register_file_handler()` - Add file handlers for layer processing for template processing

static ksconf_cli_modify_argparse(*parser: Any*, *name: str*)

Manipulate argparse rules. This could be used to add additional CLI options for other hook-added features added features

Note that this hook is called for both the top-level argparse instance as well as each subparser. The `name` argument should be inspected to determine if the parse instances is the parent (top-level) parser, or some other named subcommands.

static ksconf_cli_process_args(*args: Any*)

Hook to capture all parsed arguments, includes any custom arguments added to the CLI via the the `ksconf_cli_modify_argparse()` hook. `args` can be mutated directly, if needed.

static modify_jinja_env(*env: Any*)

Modify the Jinja2 environment object. This can be used to add custom filters or tests, for example.

Invoked by `LayerFile_Jinja2` immediately after initial Environment creation. `env` should be mutated in place.

static package_pre_archive(*app_dir: Path*, *app_name: str*)

Modify, inventory, or test the contents of an app before the final packaging commands. This can be triggered from the `ksconf package` command or via the API.

During a `ksconf package` process, this hook executes right before the final archive is created. All local merging, app version or build updates, and so on are completed before this hook is executed.

From an API perspective, this hook is called from `ksconf.package.AppPackager` whenever a content freeze occurs, which is typically when `make_archive()` or `make_manifest()` is invoked.

static post_combine(*target: Path*, *usage: str*)

Trigger a custom action after a layer combining operation. This is used by multiple ksconf subcommands and the API.

This trigger could be used to modify the file system, trigger external operations, track/audit behaviors, and so on.

When using CLI commands, `usage` should be either "combine" or "package" depending on which ksconf command was invoked. Direct invocation of `LayerCombiner` can pass along a custom usage label and avoid impacting CLI, when desirable.

If your goal is to only trigger an action during the app packaging process, also consider the `package_pre_archive()` hook, which may be more appropriate.

ksconf.layer module

class ksconf.layer.**DirectLayerRoot**(*context:* LayerContext = *None*)

Bases: LayerRootBase

A very simple direct LayerRoot implementation that relies on all layer paths to be explicitly given without any automatic detection mechanisms. You can think of this as the legacy implementation.

add_layer(*path: Path*)

order_layers()

class ksconf.layer.**DotDLayerRoot**(*context=None*)

Bases: LayerRootBase

class Layer(*name: str*, *root: Path*, *physical: PurePath*, *logical: PurePath*, *context:* LayerContext, *file_factory: Callable*, *prune_points: set[Path] = None*)

Bases: Layer

prune_points: set[Path]

walk() → Iterator[tuple[Path, list[str], list[str]]]

apply_filter(*layer_filter:* LayerFilter)

Apply a destructive filter to all layers. `layer_filter(layer)` will be called one for each layer, if the filter returns True than the layer is kept. Root layers are always kept.

Returns True if layers were removed

layer_regex = re.compile('(?P<layer>\\d\\d-[\\w_.-]+)')

list_layers() → List[*Layer*]

mount_regex = re.compile('(?P<realname>[\\w_.-]+)\\.d$')

order_layers()

set_root(*root: Path*, *follow_symlinks=None*)

Set a root path, and auto discover all '.d' directories.

Note: We currently only support `.d/<layer>` directories, a file like `default.d/10-props.conf` won't be handled here. A valid name would be `default.d/10-name/props.conf`.

class ksconf.layer.**FileFactory**

Bases: object

disable(*name*)

enable(*name*, *_enabled=True*)

list_available_handlers() → list[str]

register_handler(*name: str*, ***kwargs*)

class ksconf.layer.**LayerContext**(*follow_symlink: 'bool' = False*, *block_files: 'Match' = re.compile('\\.(bak|swp)$')*, *block_dirs: 'set' = <factory>*, *template_variables: 'dict' = <factory>*)

Bases: object

block_dirs: set

block_files: Match = re.compile('\\.(bak|swp)$')

follow_symlink: bool = False

template_variables: dict

exception ksconf.layer.**LayerException**

Bases: Exception

class ksconf.layer.**LayerFile**(*layer:* Layer, *relative_path: PurePath*, *stat: stat_result = None*)

Bases: PathLike

Abstraction of a file within a Layer

Path definitions

logical_path
: Conceptual file path. This is the final path after all layers are resolved. Think of this as the 'destination' file.

physical_path
: Actual file path. The location of the physical file found within a source layer. Most of the time this is the 'source' file, however this doesn't take into considerations layer combining or template expansion requirements. (In the case of a template, this would be the template file)

resource_path
: Content location. Often this the physical_path, but in the case of abstracted layers (like templates, or archived layers), this would be the location of a temporary resource that contains the expanded/rendered content.

layer

property logical_path: Path

static match(*path: PurePath*)

property mtime

property physical_path: Path

relative_path

property resource_path: Path

property size

property stat: stat_result

class ksconf.layer.**LayerFile_Jinja2**(**args*, ***kwargs*)

Bases: LayerRenderedFile

property jinja2_env

static match(*path: PurePath*)

render(*template_path: Path*) → str

static transform_name(*path: PurePath*)

class ksconf.layer.**LayerFilter**

Bases: object

add_rule(*action*, *pattern*)

evaluate(*layer:* Layer) → bool

class ksconf.layer.**LayerRenderedFile**(**args*, ***kwargs*)

Bases: LayerFile

Abstract LayerFile for rendered scenarios, such as template scenarios. A subclass really only needs to implement `match()` `render()`

property logical_path: Path

property physical_path: Path

render(*template_path: Path*) → str

property resource_path: Path

static transform_name(*path: PurePath*)

use_secure_delete = False

class ksconf.layer.**LayerRootBase**(*context:* LayerContext = *None*)

Bases: object

All 'path's here are relative to the ROOT.

class Layer(*name: str*, *root: Path*, *physical: PurePath*, *logical: PurePath*, *context:* LayerContext, *file_factory: Callable*)

Bases: object

Basic layer Container: Connects logical and physical paths.

context

get_file(*path: Path*) → *LayerFile*

Return file object (by logical path), if it exists in this layer.

iter_files() → Iterator[*LayerFile*]

list_files() → list[*LayerFile*]

logical_path

name

physical_path

root

walk() → Iterator[tuple[Path, list[str], list[str]]]

add_layer(*layer:* Layer, *do_sort=True*)

apply_filter(*layer_filter:* LayerFilter) → bool

Apply a destructive filter to all layers. `layer_filter(layer)` will be called one for each layer, if the filter returns True than the layer is kept. Root layers are always kept.

Returns True if layers were removed

get_file(*path*) → Iterator[*LayerFile*]

return all layers associated with the given relative path.

get_layers_by_name(*name: str*) → Iterator[*Layer*]

iter_all_files() → Iterator[*LayerFile*]

Iterator over all physical files.

list_files() → list[*LayerFile*]

Return a list of logical paths.

list_layer_names() → list[str]

list_layers() → List[*Layer*]

list_logical_files() → list[*LayerFile*]

Return a list of logical paths.

list_physical_files() → list[*LayerFile*]

order_layers()

exception ksconf.layer.**LayerUsageException**

Bases: LayerException

ksconf.layer.**register_file_handler**(*name: str*, ***kwargs*)

ksconf.package module

class ksconf.package.**AppPackager**(*src_path*, *app_name: str*, *output: TextIO*, *template_variables: dict = None*, *predictable_mtime: bool = True*)

Bases: object

block_local(*report=True*)

blocklist(*patterns*)

check()

Run safety checks prior to building archive:

1. Set app name based on app.conf [package] id, if set. Otherwise confirm that the package id and top-level folder names align.
2. Check for files or directories starting with `.`, makes AppInspect very grumpy!

cleanup()

combine(*src*, *filters*, *layer_method='dir.d'*, *allow_symlink=False*)

expand_new_only(*value: str*) → str | None

Expand a variable but return False if no substitution occurred

Parameters
value (str) – String that may contain `{{variable}}` substitution.

Returns
Expanded value if variables were expanded, else False

Return type
str

expand_var(*value: str*) → str

Expand a variable, if present

Parameters
value (str) – String that main contain `{{variable}}` substitution.

Returns
Expanded value

Return type
str

freeze(*caller_name*)

Initiate a content freeze by restricting mutable methods. The "package_pre_archive" hook is invoked before freeze operation. Such hooks may choose to mutate the filesystem at `app_dir`, the only assumption is that all work is done before the hook returns.

Freeze can be safely called multiple times. `caller_name` is simply a label used in an exception message if the programmer screwed up.

make_archive(*filename*, *temp_suffix: str = '.tmp'*)

Create a compressed tarball of the build directory.

make_manifest(*calculate_hash=True*) → *AppManifest*

Create a manifest of the app's contents.

merge_local()

Find everything in local, if it has a corresponding file in default, merge.

require_active_context(*mutable=True*)

Decorator to mark member functions that cannot be used until the context manager has been activated.

update_app_conf(*version: str = None*, *build: str = None*)

Update version and/or build in `apps.conf`

class ksconf.package.**AppVarMagic**(*src_dir*, *build_dir*, *meta=None*)

Bases: `object`

A lazy loading dict-like object to fetch things like app version and such on demand.

expand(*value: str*) → str

A simple Jinja2 like `{{VAR}}` substitution mechanism.

get_app_id()

Splunk app package id from app.conf

get_build()

Splunk app build fetched from app.conf

get_git_head()

Git HEAD rev abbreviated

get_git_last_rev()

Git abbreviated rev of the last change of the app. This may not be the same as HEAD.

get_git_tag()

Git version tag using the `git describe --tags` command

get_layers_hash()

Build a unique hash representing the combination of ksconf layers used.

get_layers_list()

List of ksconf layers used.

get_version()

Splunk app version fetched from app.conf

git_single_line(**args*)

list_vars()

Return a list of (variable, description) available in this class.

exception ksconf.package.**AppVarMagicException**

Bases: KeyError

exception ksconf.package.**PackagingException**

Bases: Exception

ksconf.package.**find_conf_in_layers**(*app_dir*, *conf*, **layers*)

ksconf.package.**get_merged_conf**(*app_dir*, *conf*, **layers*)

ksconf.package.**normalize_directory_mtime**(*path*)

Walk a tree and update the directory modification times to match the newest time of the children. This results in a more predictable behavior over multiple executions.

ksconf.setup_entrypoints module

Defines all command prompt entry points for CLI actions

This is a silly hack allows for fallback mechanism when

(a) running unit tests (can happen before install)

(b) unexpected issues with importlib.metadata or backport

class ksconf.setup_entrypoints.**Ep**(*name*, *module_name*, *object_name*)

Bases: NamedTuple

property formatted

module_name: str

Alias for field number 1

name: str

Alias for field number 0

object_name: str

Alias for field number 2

class ksconf.setup_entrypoints.**LocalEntryPoint**(*data*)

Bases: object

Bare minimum stand-in for entrypoints.EntryPoint

load()

ksconf.setup_entrypoints.**debug**()

ksconf.setup_entrypoints.**get_entrypoints_fallback**(*group*)

ksconf.setup_entrypoints.**get_entrypoints_setup**()

Build entry point text descriptions for ksconf packaging

ksconf.xmlformat module

class ksconf.xmlformat.**FileReadlinesCache**

Bases: object

Silly workaround for CDATA detection...

static convert_filename(*filename*)

readlines(*filename*)

class ksconf.xmlformat.**SplunkSimpleXmlFormatter**

Bases: object

static cdata_tags(*elem: Any*, *tags: List[str]*)

Expand text to CDATA, if it isn't already.

classmethod expand_tags(*elem: Any*, *tags: set*)

Keep <elem></elem> instead of shortening to <elem/>

classmethod format_json(*elem: Any*, *indent=2*)

Format JSON data within a Dashboard Studio dashboard. This is still pretty limited (for example, long searches still show up on a single line), but this give you at least a fighting change to figure out what's different.

classmethod format_xml(*src*, *dest*, *default_indent=2*)

static guess_indent(*elem: Any*, *default=2*)

classmethod indent_tree(*elem: Any*, *level=0*, *indent=2*)

keep_tags = {'default', 'earliest', 'fieldset', 'label', 'latest', 'option', 'search', 'set'}

Build example

Take a look at this example `build.py` file that use the `ksconf.builder` module.

```
#!/usr/bin/env python
#
# KSCONF Official example app building script
#
# NOTE:  Keep in mind that this is all very experimental and subject to change.
import sys
from pathlib import Path

from ksconf.builder import QUIET, VERBOSE, BuildManager, BuildStep, default_cli
from ksconf.builder.steps import clean_build, copy_files, pip_install

```

(continues on next page)

(continued from previous page)

```
manager = BuildManager()

APP_FOLDER = "TA-my_technology"
SPL_NAME = "ta_my_technology-{{version}}.tgz"
SOURCE_DIR = "."

REQUIREMENTS = "requirements.txt"

# Files that support the build process, but don't end up in the tarball.
BUILD_FILES = [
    REQUIREMENTS,
]

COPY_FILES = [
    "README.md",
    "bin/*.py",
    "default/",
    "metadata/*.meta",
    "static/",
    "lookups/*.csv",
    "appserver/",
    "README/*.spec",
] + BUILD_FILES


@manager.cache([REQUIREMENTS], ["lib/"], timeout=7200)
def python_packages(step):
    # Reuse shared function from ksconf.build.steps
    pip_install(step, REQUIREMENTS, "lib",
                handle_dist_info="remove")


def package_spl(step: BuildStep):
    log = step.get_logger()
    top_dir = step.dist_path.parent
    release_path = top_dir / ".release_path"
    release_name = top_dir / ".release_name"
    # Verbose message
    log("Starting to package SPL file!", VERBOSE)
    step.run(sys.executable, "-m", "ksconf", "package",
             "--file", step.dist_path / SPL_NAME,   # Path to created tarball
             "--app-name", APP_FOLDER,              # Top-level directory name
             "--block-local",                       # VC build, no 'local' folder
             "--release-file", str(release_path),
             ".")
    # Provide the dist file as a short name too (used by some CI/CD tools)
```

(continues on next page)

(continued from previous page)

```
58      path = release_path.read_text()
59      short_name = Path(path).name
60      release_name.write_text(short_name)
61      # Output message will be produced even in QUIET mode
62      log(f"Created SPL file:  {short_name}", QUIET)
63
64
65  def build(step: BuildStep, args):
66      """ Build process """
67      # Step 1:  Clean/create build folder
68      clean_build(step)
69
70      # Step 2:  Copy files from source to build folder
71      copy_files(step, COPY_FILES)
72
73      # Step 3:  Install Python package dependencies
74      python_packages(step)
75
76      # Step 4:  Make tarball
77      package_spl(step)
78
79
80  if __name__ == '__main__':
81      # Tell build manager where stuff lives
82      manager.set_folders(SOURCE_DIR, "build", "dist")
83
84      # Launch build CLI
85      default_cli(manager, build)
```

Usage notes

- `BuildManager` - is used to help orchestrate the build process.
- `step` is an instance of `BuildStep`, which is passed as the first argument to all the of step-service functions. This class assists with logging, and directing all activities to the correct paths.
- There's no interal interface for *ksconf package* yet, hence another instance of Python is launched on line 48. This is done using the module execution mode of Python, which is a slightly more reliable way of launching ksconf from within itself. For whatever that's worth.

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[SPLKDOC1] https://docs.splunk.com/Documentation/Splunk/7.2.3/Admin/Configurationfiledirectories

PYTHON MODULE INDEX

k

INDEX

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X