Extending SaltStack

Salt already ships with a very powerful set of tools, but that doesn’t mean that they all suit your needs perfectly. Extending SaltStack follows a tutorial-based approach to explain different types of modules, from fundamentals to complete and fully functioning modules.

Starting with the Loader system that drives Salt, this book will guide you through the most common types of modules. Next, you will learn how to write execution modules and extend the configuration using the grain, pillar, and SDB modules. Soon after, you will be taken through the state modules and then the renderers that can be used with them. This will be followed by returner and output modules. After that, there will be modules for external file servers, clouds, beacons, and finally external authentication and wheel modules to manage the master.

With this guide in hand, you will be prepared to create, troubleshoot, and manage the most common types of Salt modules and take your infrastructure to new heights!

Who this book is written for
This book is for both new and existing Salt developers who are looking to build and write new Salt modules. Some prior Python development experience is expected.

What you will learn from this book
- Understand the working of Salt’s Loader system
- Write several of the most common types of Salt modules
- Interact with different kinds of modules and build new ones
- Submit open source modules upstream to the Salt project
- Make Salt interact with third-party services and applications

Extending SaltStack

Extend the power of your infrastructure and applications with Salt modules

Foreword by Eric Johnson - Author of SaltStack’s Google Compute Engine cloud module

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 5 'Rendering Data'
- A synopsis of the book’s content
- More information on Extending SaltStack
Joseph Hall has been working with SaltStack for a very long time. His first commit was on March 14, 2011, making him the second contributor to the Salt codebase. At the time his Python skills weren't very good, but writing Salt modules made them better. He has written a number of Salt modules and is planning to write many more. He has also written Mastering SaltStack, Packt Publishing.
You hold in your hands (or in your e-reader) the first book dedicated to writing code to be used with the SaltStack framework of tools.

**What this book covers**

*Chapter 1, Starting with the Basics,* starts with a discussion of the two core principles that this book focuses on: how Salt uses Python and how the Loader system works. These form the foundation of extending Salt.

*Chapter 2, Writing Execution Modules,* explains that the heavy lifting in most of Salt is performed by execution modules, which are often wrapped by other modules. A solid understanding of execution modules will also be key to understanding how other module types work.

*Chapter 3, Extending Salt Configuration,* explains that the ability to dynamically manage configuration can make some modules far more useful. Some modules won't even work without dynamic configuration. Here, we explore different ways to provide that.

*Chapter 4, Wrapping States Around Modules,* supports the fact that execution modules make things work, but state modules make that work persist. In this chapter, you will see how to manage execution modules using state modules.

*Chapter 5, Rendering Data,* shows that the renderer system allows you to add your own templating systems, increasing the power of states. Jinja and YAML are all well and good, but sometimes, you need something more.

*Chapter 6, Handling Return Data,* answers the query what happens to the data when a job finishes. There are many places it can go, and you can write modules to send it there.
Chapter 7, Scripting with Runners, shows that SaltStack knows that system administrators have used scripting languages for years, and they have provided a scripting environment that combines Python with the raw power of Salt.

Chapter 8, Adding External File Servers, advises not to just serve files from Salt Master. You can serve files from wherever you want with your own external file server module.

Chapter 9, Connecting to the Cloud, helps you find out how you can update existing cloud modules or add your own. Everyone uses the cloud now, and Salt Cloud connects it to Salt.

Chapter 10, Monitoring with Beacons, helps us to solve the problem that Salt isn't normally associated with monitoring, which is a shame. Beacons are one way to integrate Salt into your monitoring framework.

Chapter 11, Extending the Master, explains that Salt provides a way for you to serve the administrative needs of the Master programmatically. Bonus points for tying in your own authentication system to Salt.

Appendix A, Connecting Different Modules, gives solutions to how to fit the different components even if it is known that Salt modules are designed to play together. This appendix lays out how the different parts connect together.

Appendix B, Contributing Code Upstream, gives you tips to know where the project is screwed up or what features are missing. It doesn't have to be that way with Salt but going back to the community.
Having the ability to write your own execution and state modules is powerful from a developer's point of view, but you cannot overlook being able to provide that kind of power to users who do not have the ability to provide modules of their own.

Renderers allow users to provide data to various parts of Salt using different kinds of data input formats. The handful of renderers that ship with Salt cover the majority of use cases, but what if your users need to apply data in a specialized format? Or even a more common one that is not yet supported, such as XML? In this chapter, we'll discuss:

- Writing renderers
- Troubleshooting renderers

Understanding file formats

By default, Salt uses YAML for its various files. There are two primary reasons for this:

- YAML is easily converted into Python data structures
- YAML is easy for humans to read and modify

Salt configuration files must be in YAML as well (or JSON, which can be read by YAML parsers), but other files such as states, pillars, reactors, and so on can use other formats. A data serialization format is the most common, but any format that can be translated into a Python dictionary will do just fine.

For example, there are three different Python renderers that ship with Salt: `py`, `pyobjects`, and `pydsl`. Each has its strengths and weaknesses, but the end result is the same: they execute Python code that results in a dictionary, which is then passed into Salt.
Generally speaking, you will find two types of renderers inside of Salt. The first returns data in a Python data structure. Both serializers and code-based modules fit into this category. The second is for managing text formatting and templating. Let's talk about each in turn, and then build our own renderers later on in the chapter.

**Serializing data**
Data can be stored in any number of formats, but in the end, that data must be something that can be turned into instructions. Formats such as YAML and JSON are obvious choices, because they are easy to modify and mirror the resulting data structures in the program that uses them. Binary formats such as Message Pack aren't as easily modified by humans, but they still result in the same data structures.

Other formats, such as XML, are more difficult because they don't directly resemble the internal data structures of programs like Salt. They're great for modeling code that makes heavy use of classes, but Salt doesn't make much use of such code. However, when you know how such a format can be converted into a data structure that Salt can use, then building a renderer for it is not difficult.

**Working with templates**
Templates are important because they allow end users to use certain programmatic elements without having to write actual modules. Variables are certainly one of the most critical elements of a templating engine, but other constructs such as loops and branching can also give a lot of power to the user.

Templating renderers differ from data-serializing renderers in that instead of returning data in a dictionary format, which is then ingested by Salt, they return data that must be converted at least one more time, using a data-serialization renderer.

This may seem counterintuitive on some levels, but the use of render pipes brings these two elements together.

**Using render pipes**
Render pipes are based on Unix pipes; data can be passed from module to module through a series of pipes, in order to arrive at the final data structure. You may not realize it, but if you've ever written an SLS file, you've used a render pipe.
To set up a render pipe, you add a line to the top of the file to be rendered, which contains the classic Unix hashbang, followed by the renderers to be used, in the order to be used, separated by the pipe character. The default rendering sequence is effectively:

```#!jinja|yaml```

This means that the file in question will be first parsed by Jinja2, and compiled into a format that can be read by the YAML library.

It's generally not reasonable or necessary to pipe more than two different renderers together; the more that are used, the more complicated the resulting file is to understand by humans, and the greater the chance for errors. Generally, a templating engine that adds programmatic shortcuts, and a data serializer, is plenty. One notable exception is the gpg renderer, which can be used for encryption-at-rest scenarios. The hashbang for this would look like:

```#!jinja|yaml|gpg```

### Building a serializing renderer

Renderers are reasonably easy to build, because they typically do little more than import a library, shove data through it, and then return the result. Our example renderer will make use of Python's own Pickle format.

#### The basic structure

Outside of any necessary imports, a renderer requires only a `render()` function. The most important argument is the first. As with other modules, the name of this argument is not important to Salt, so long as it is defined. Because our example uses the pickle library, we'll use `pickle_data` as our argument name.

Other arguments are also passed into renderers, but in our case we'll only use them for troubleshooting. In particular, we need to accept `saltenv` and `sls`, with the defaults shown later. We'll cover those in the *Troubleshooting Renderers* section, but for now we'll just use `kwargs` to cover them.

We also need to start with a special kind of import, called `absolute_import`, that allows us to import the pickle library from a file that's also called pickle.
Let’s go ahead and lay out the module, and then talk about the components in the `render()` function:

```python
from __future__ import absolute_import
import pickle
from salt.ext.six import string_types

def render(pickle_data, saltenv='base', sls='', **kwargs):
    """
    Accepts a pickle, and renders said data back to a python dict.
    """
    if not isinstance(pickle_data, string_types):
        pickle_data = pickle_data.read()

    if pickle_data.startswith('#!'):
        pickle_data = pickle_data[(pickle_data.find('
') + 1):]
    if not pickle_data.strip():
        return {}
    return pickle.loads(pickle_data)
```

This function does not do much, other than:

- First, check to see whether the data being passed in is a string, and if not, treat it as a file-like object.
- Check for the existence of a #!, indicating the use of an explicit render pipe. Because that pipe is handled elsewhere, and it will cause errors with the pickle library, discard it.
- Check to see whether the resulting content is empty. If so, return an empty dictionary.
- Run the data through the pickle library, and return the result.

If you start comparing this code with the renderers that ship with Salt, you’ll find that many of them are almost identical. This is in part because so many data serialization libraries in Python use exactly the same methods.
Let's put together a file that can be used. The example data that we'll use is:

```python
apache:
  pkg:
    - installed
    - refresh: True
```

The best way to create this file is with Python itself. Go ahead and open up a Python shell and type the following commands:

```python
>>> import pickle
>>> data = {'apache': {'pkg': ['installed', {'refresh': True}]}
>>> out = open('/srv/salt/pickle.sls', 'w')
>>> pickle.dump(data, out)
>>> out.close()
```

When you exit out of the Python shell, you should be able to open this file in your favorite text editor. When you add a hashbang line to the top that specifies the `pickle` renderer, your file will probably look like this:

```plaintext
#!/pickle
(dp0
 S'apache'
p1
(dp2
 S'pkg'
p3
 (lp4
 S'installed'
p5
 a(dp6
 S'refresh'
p7
 I01
 sass.
```

Save the file, and use `salt-call` to test out your renderer. This time, we'll tell Salt to dump out the resulting SLS, as Salt sees it:

```
# salt-call --local state.show_sls pickle --out=yaml
local:
  apache:
    __env__: base
```
Salt's state compiler adds some extra information that it uses internally, but we can see that the basics of what we requested are there.

## Building a templating renderer

Building a renderer that handles templating files is not that different from one that does serialization. In fact, the renderer itself is pretty much the same, outside of the library-specific code. This time, we'll use a Python library called tenjin. You may need to install it using pip:

```
# pip install tenjin
```

## Templating with Tenjin

This module makes use of a third-party library, so there will be a `__virtual__()` function to make sure it's installed:

```python
...  
Conver a file using the Tenjin templating engine

This file should be saved as salt/renderers/tenjin.py
...
from __future__ import absolute_import
try:
    import tenjin
    from tenjin.helpers import *
    HAS_LIBS = True
except ImportError:
    HAS_LIBS = False
from salt.ext.six import string_types

def __virtual__():
    ...
    Only load if Tenjin is installed
    ...
```
return HAS_LIBS

def render(tenjin_data, saltenv='base', sls='', **kwargs):
    '''
    Accepts a tenjin, and renders said data back to a python dict.
    '''
    if not isinstance(tenjin_data, string_types):
        tenjin_data = tenjin_data.read()

    if tenjin_data.startswith('#!'):
        tenjin_data = tenjin_data[(tenjin_data.find('
') + 1):]
    if not tenjin_data.strip():
        return {}

    template = tenjin.Template(input=tenjin_data)
    return template.render(kwargs)

The render() function itself is fundamentally identical to the one that we used for pickle, except for the last two lines, which handles the templating engine slightly differently.

Take note of the kwargs that are passed into this function. Templating engines generally have the ability to merge in an external data structure, which can be used with the various data structures in the templating engine itself. Salt will make some data available inside kwargs, so we'll pass that in for Tenjin to use.

**Using a templating renderer**

Of course, you'll need a hashbang line in your SLS files as before, but since our Tenjin renderer isn't set up to return straight data, you will need to add the name of the desired data-serialization renderer to your render pipe. We'll use the same actual SLS data as before, but with a couple of Tenjin-specific elements added:

```yaml
#!tenjin|yaml
<?py pkg = 'apache'>
<?py refresh = True?>
#{pkg}:
    pkg:
        - installed
        - refresh: #{refresh}
```
Rendering Data

We haven't done anything special here, just set a couple of variables, and then used them. The resulting content will be in YAML format, so we've added `yaml` to our render pipe.

A number of templating engines, including Tenjin, have the ability to process templates that output either strings (as we've done in our example), or an actual data structure, such as what a data serializer would return. When using such a library, take a moment to consider how much of it you plan to use, and whether it makes sense to create two distinct renderers for it: one for data and one for strings.

Testing is much the same as before:

```sh
# salt-call --local state.show_sls tenjin --out yaml
local:
  apache:
    pkg:
      - installed
      - refresh: true
      - order: 10000
    __sls__: !!python/unicode tenjin
    __env__: base
```

We can see slight differences between our first example and our second, but those differences just show which module was used to render the data.

Troubleshooting renderers

Because renderers are so often used to manage SLS files, it is often easiest to troubleshoot them using the state compiler, as we have been doing already in this chapter.

First, generate a small SLS file that contains the specific elements which you need to test. This will either be a data file in the format that a serialization engine uses, or a text-based file that results in a data-serialization file format. If you are writing a templating renderer, it is often easiest to just use YAML.
The state execution module contains a number of functions that exist primarily for troubleshooting. We used state.show_sls in our examples, with --out yaml, because it displays the output in a format that we're already used to in our SLS files. However, some other useful functions are:

- **state.show_low_sls**: Shows data from a single SLS file, after it has been converted to low data by the State compiler. Low data is often easier to visualize when writing state modules.
- **state.show_highstate**: Shows all of the states, as they would be applied to a Minion, according to the top.sls file. The output from this will look as if all of the SLS files have been shoved together. This can be useful when troubleshooting rendering issues that you believe span across multiple SLS files.
- **state.show_lowstate**: The data returned from this function is the same as what state.show_highstate returns, but after being processed by the state compiler. Again, this is like a long version of state.show_low_sls.

**Summary**

Renderers are used to convert various file formats into a data structure that is usable internally by Salt. Data-serialization renderers return data in a dictionary format, whereas templating renderers return data that can be processed by a data serializer. Both types of renderer look the same, and require a render() function.

Now that we know how to handle the data going into Salt, it's time to look at the data coming back out of Salt. Next up: handling return data.
Where to buy this book
You can buy Extending SaltStack from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.