idem Documentation

VMware, Inc.

Nov 16, 2022

CONTENTS

CHAPTER

ONE

IDEM

Idem is an idempotent dataflow programming language. It exposes stateful programming constructs that makes things like enforcing the state of an application, configuration, SaaS system, or others very simple.

Since Idem is a programming language, it can also be used for data processing and pipelining. Idem can be used not only to manage the configuration of interfaces, but also for complex rule engines and processing files or workflows.

Idem is a language to glue together management of all sorts of interfaces. You can think of it like having idempotent scripts. Automation that can be run over and over again that enforces a specific state or process.

Idem is unique in that it is built purely as a language. It can be added to any type of management system out there and can be applied in a cross platform way easily.

Idem's functionality can also be expanded easily. Instead of storing all of the language components in a single place, the libraries used by Idem can be written independently and seamlessly merged into Idem, just like a normal programming language!

1.1 What does Idempotent mean?

The concept of Idempotent is simple! It just means that every time something is run, it always has the same end result regardless of the state of a system when the run starts!

At first glance this might seem useless, but think more deeply. Have you ever needed to make sure that something was set up in a consistent way? It can be very nice to be able to enforce that setup without worrying about breaking it. Or think about data pipelines, have you ever had input data that needed to be processed? Idempotent systems allow for data to be easily processed in a consistent way, over and over again!

1.2 How Does This Language Work?

Idem is built using two critical technologies, *Python* and *POP*. Since Idem is built on Python it should be easy to extend for most software developers. Extending Idem can be very easy because simple Python modules are all you need to add capabilities!

The other technology, *POP*, may be new to you. This is the truly secret sauce behind Idem as well as a number of emerging exciting technologies. *POP* stands for Plugin Oriented Programming. It is the brainchild of the creator of Salt and a new way to write software. The *POP* system makes the creation of higher level paradigms like Idem possible, but also provides the needed components to make Idem extensible and flexible. If *POP* is a new concept to you, check it out!

Idem works by taking language files called *sls* files and compiling them down to data instructions. These data instructions are then run through the Idem runtime. These instructions inform Idem what routines to call to enforce state or process data. It allows you to take a high level dataset as your input, making the use of the system very easy.

1.3 Paradigms and Languages, This Sounds Complicated!

Under the hood, it is complicated! The guts of a programming language are complicated, but it is all there to make your life easier! You don't need to understand complex computer science theory to benefit from Idem. You just need to learn a few simple things and you can start making your life easier today!

CHAPTER

TWO

CONFIG TEMPLATE

To save all of your CLI flags in a single config file, run the full idem CLI command that you want, with `--config-template` as an additional flag. The generated config file will also include options for plugins in adjacent projects that are not necessarily exposed in the idem CLI command.

At the time of this writing, a config template will include the settings shown in the following example. Note that some settings are empty because the example command only included `--config-template` and no other flags.

```
$ idem --config-template
```

Resulting config file:

```
acct:
  acct_file:
  acct_key:
  crypto_plugin: fernet
  extras:
  output_file: null
  serial_plugin: msgpack
  allowed_backend_profiles:
  render_pipe: jinja|yaml
evbus:
  serial_plugin: json
idem:
  acct_profile: default
  cache_dir: ~/.idem/var/cache/idem
  esm_keep_cache: false
  esm_plugin: local
  esm_profile: default
  esm_serial_plugin: msgpack
  exec: ''
  exec_args: []
  log_datefmt: '%H:%M:%S'
  log_file: idem.log
  log_fmt_console: '[%(levelname)-8s] %(message)s'
  log_fmt_logfile: '%(asctime)s,%(msecs)03d [%(name)-17s][%(levelname)-8s] %(message)s'
  log_handler_options:
  log_level: warning
  log_plugin: basic
  param_sources: []
  params: ''
  pending: default
```

(continues on next page)

(continued from previous page)

```
  reconciler: none
  render: jinja|yaml|replacements
  root_dir: ~/.idem
  run_name: cli
  runtime: parallel
  sls: []
  sls_sources: []
  test: false
  tree: ''
pop_config:
  log_datefmt: '%H:%M:%S'
  log_file: idem.log
  log_fmt_console: '[%(levelname)-8s] %(message)s'
  log_fmt_logfile: '%(asctime)s,%(msecs)03d [%(name)-17s][%(levelname)-8s] %(message)s'
  log_handler_options: *id001
  log_level: warning
  log_plugin: basic
rend:
  file: null
  output: null
  pipe: yaml
```

To run an idem command that uses the settings from a config file, add the `--config` option. The following `idem state` example has `my_config.cfg` as the saved config file.

```
$ idem state --config=my_config.cfg
```

CHAPTER

THREE

IDEM DOC

The `idem doc` subcommand has prints function documentation for references in the code. It will return all metadata about a function reference on the hub. It uses the pop-tree project under the hood to parse references on the hub.

Running the `idem doc` subcommand will return all references that match the given reference.

```
idem doc <function reference on the hub>
```

Each returned reference will contain the following values.

3.1 doc

The function docstring.

3.2 file

The file that owns this particular reference on the hub.

3.3 start_line_number

The line number in the file where the function begins, useful for pyls-pop.

3.4 end_line_number

The line number in the file where the function ends, useful for pyls-pop.

3.5 ref

The reference to this function on the hub

3.6 contracts

There are three kinds of contracts, each function lists the references to all contracts that it implements.

- pre: A list of the function's `pre` contracts
- post: A list of the function's `post` contracts
- call: A list containing a function's `call` contract

3.7 parameters

Each parameter in the function header is listed. There are two possible values for a parameter:

- default: If this key is present for a parameter, it contains it's default value
- annotation: The typehint for a parameter if one exists

3.7.1 Examples

Get the documentation from a specific exec module function.

```
idem doc exec.test.ping
```

output:

```
exec.test.ping:
----------
doc:
    Immediately return success
file:
    ~/PycharmProjects/idem/idem/exec/test.py
start_line_number:
    13
end_line_number:
    15
ref:
    exec.test.ping
contracts:
    ----------
    pre:
    call:
        - exec.recursive_contracts.soft_fail.call
    post:
        - exec.recursive_contracts.init.post
parameters:
    ----------
```

(continues on next page)

(continued from previous page)

```
    hub:
        ----------
```

Get the documentation for a specific state module function.

```
idem doc states.test.present
```

output:

```
states.test.present:
----------
doc:
    Return the previous old_state and the given new_state.
    Raise an error on fail
file:
    ~/PycharmProjects/idem/idem/states/test.py
start_line_number:
    279
end_line_number:
    295
ref:
    states.test.present
contracts:
    ----------
    pre:
        - states.recursive_contracts.init.pre
    call:
    post:
        - states.recursive_contracts.resource.post_present
        - states.recursive_contracts.init.post
parameters:
    ----------
    hub:
        ----------
    ctx:
        ----------
    name:
        ----------
        annotation:
            <class 'str'>
    new_state:
        ----------
        default:
            None
    result:
        ----------
        default:
            True
    force_save:
        ----------
        default:
            None
```

Return all functions in a single module:

```
idem doc states.aws.ec2.vpc
```

Return all functions in a sub:

```
idem doc states.aws
```

Return absolutely every reference on the hub:

```
idem doc
```

CHAPTER

FOUR

EXTENDING IDEM

Extending Idem is simple, but it does require a few steps. To extend Idem you need to create a new Idem plugin project using *POP*. Now don't run away, this has been designed to be easy!

4.1 What is POP?

You don't need to understand the inner workings of Plugin Oriented Programming or *pop* to extend Idem, just think about it as a system for writing and managing plugins. Idem is all about plugins!

If you want to learn more about the details of *POP*, take a look at the docs. It is powerful stuff and might change how you program forever: https://pop.readthedocs.io

4.2 Lets Get Down to Business

Start by installing *idem*:

```
pip install idem
```

This will download and install both *idem* and *pop*. Now you can start your project by calling *pop-create* to make the structure you need:

```
pop-create idem_tester -t v -d exec states
```

By passing *-t v* to *pop-create* we are telling *pop-create* that this is a *Vertical App Merge* project. By passing *-d exec states* we are asking *pop-create* to add the 2 dynamic names *exec* and *states* to the project.

This will create a new project called *idem_tester* with everything you need to get the ball rolling.

4.3 Making Your First Idem State

In your new project there will be a directory called *idem_tester/states*, in this directory add a file called *trial.py*:

```
async def run(hub, ctx, name):
    """
    Do a simple trial run
    """
    return {
        "name": name,
```

(continues on next page)

(continued from previous page)

```
        "result": True,
        "changes": {},
        "comment": "It Ran!",
    }
```

For idem to run, *states* functions need to return a python dict that has 4 fields, *name*, *result*, *changes*, and *comment*. These fields are used by Idem to not only expose data to the user, but also to track the internal execution of the system.

Next install your new project. For *idem* to be able to use it your project, it needs to be in the python path. There are a lot of convenient ways to manage the installation and deployment of *POP* projects, but for now we can just use good old *pip*:

```
pip install -e /path/to/your/project/root
```

Now you can execute a state with *idem*. As you will see, *pop* and *idem* are all about hierarchical code. *Idem* runs code out of a directory, you need to point *idem* to a directory that contains *sls* files. Go ahead and *cd* to another directory and make a new *sls* directory.

```
mkdir try
cd try
```

Now open a file called *try.sls*:

```
try something:
    trial.run
```

Now from that directory run idem:

```
idem --sls try
```

And you will see the results from running your trial.run state!

CHAPTER

FIVE

ADDING REQUISITES

Requisites are a basic language feature of idem, they allow for a definition of how to perform a dependency check. The simplest requisite is *require*. The *require* requisite simply mandates that the required ref has been processed and returned a result of *True*.

There are 2 plugin subsystems that are used in the resolution of requisites. The *idem.req* sub and the *idem.rules* sub. These are simple subs that allow the definition of the requisite as well as the rules that create the requisite check.

An *idem.req* plugin has a function called *define* which is used to define what rules are run by the requisite and how those rules are checked. This allows the rules to be re-usable for multiple requsite definitions. The simple example of the *require* requisite is this:

```
def define(hub):
    """
    Return the definition used by the runtime to insert the conditions of the
    given requisite
    """
    return {
        "result": [True, None],
    }
```

This return states that only one rule needs to be checked, the *result* rule. The value, in this case, that is passed to the result rule is the list *[True, None]*. This value is loaded into the *result* rule as the *condition* argument.

The rules are a little more complicated. They need to do the work to verify that for all of the required ID refs that the rule is followed. Here is the *result* rule:

```
def check(hub, condition, reqret, chunk):
    """
    Check to see if the result is True
    """
    if isinstance(condition, list):
        if reqret["ret"]["result"] in condition:
            return {}
    if reqret["ret"]["result"] is condition:
        return {}
    else:
        return {
            "errors": [
                f'Result of require {reqret["r_tag"]} is "{reqret["ret"]["result"]}",␣
↪not "{condition}"'
            ]
        }
```

A rule plugin needs a function called *check* that is used to check the rule. It takes 3 arguments. These arguments are *condition*, *reqret*, and *chunk*. These are objects that are internal to idem. The *condition* is the data passed in by the *define* function in the *idem.req* plugin. The *reqret* is the return information for a single required ID that has already been executed. The *chunk* is the ID declaration with the assigned requisite. The return from this function defines

CHAPTER

SIX

SLS METADATA

Sometimes it may be desirable for metadata to be stored inside of an SLS file. This can be useful for defining any additional data that an external system may want to use that is not included inside of the Idem runtime.

6.1 SLS Level Metadata

Add metadata to an SLS file is very simple, just make a top level key in the SLS file called "META":

```
META:
  foo: bar
  baz:
    - 1
    - True
    - "a string"
```

The "META" key is transferred into the idem run's running dict under the name "meta" and can be retrieved by anyone who has access to the run on the hub.

Found in *hub.idem.RUNS[<run name>]["meta"]["SLS"]*

The metadata is stored relative to the SLS reference where the original metadata was found.

6.2 ID Level Metadata

Metadata can also be stored inside the ID Declarations, this allows for metadata to be associated with an ID instead of just with the SLS file. Simply create a "META" key inside the ID Declaration:

```
private_network:
  META:
    foo: bar
  cloud.vpc:
    - cidir: 10.0.0.0/16
```

CHAPTER

SEVEN

SLS STRUCTURE

Idem utilize a system called SLS - Structured Layered States. The SLS system allows for a specific data structure that represents the desired state of a system. That target data structure can be obtained through a layered rendering process. Hence the name - Structured Layered State.

This allows for data to be represented in any way imaginable - JSON, YAML, XML, or even programming languages. This major benefit makes it easy to write Idem code in whatever way works best for you!

7.1 Core Components

This document is all about defining the core components of the SLS file, that way you can identify what the underlying data structure looks like and how to best get there. By default SLS files are represented as YAML, unlike other YAML systems you may be familiar with, the SLS format has a finite dept, making it very easy to learn, read, and write.

The first components we will discuss are the *ID Declaration*, *Path Reference*, and *Arguments*. The core use of all SLS files can be encapsulated inside these three simple components:

```
Some_Desired_State:  # ID Declaration
  cloud.instance.present:  # Path Reference
    - option: value  # Argument
```

7.1.1 ID Declaration

The ID Declaration defines the top level identifier user for all reverences under it. The ID Declaration is also passed to the state function as the *name*, unless an argument is passed as *name* under the *Path Reference*.

7.1.2 Path Reference

The *Path Reference* specifies what underlying function is being called to enforce the idempotent state for the target cloud/API/system. The dot delimited *Path Reference* links directly to how plugins are loaded into Idem using POP. The *Path Reference* is a literal reference to a location on the *hub* inside of Idem. The *hub* contains all of the code that Idem runs, therefore the *Path Reference* is a literal path to the code location translated as *hub.idem.states.*<*Path Reference*>.

The main benefit here is that the *Path Reference* gives you a direct insight into where the code that is being called resides. This makes development and debugging very simple. If a code issue exists with a state in Idem then you will know just where to find it!

Path Components

The path is broken up into two components, the *state ref*, and the *function ref*. The periods delineate the two references. Everything after the last period is the *function ref* and everything before the last period is the *state ref*.

For instance, if the *Path Ref* is *cloud.network.present*, then the *state ref* is *cloud.network* and the *function ref* is *present*.

7.1.3 Arguments

Since the *Path Reference* is a path to a function, the arguments are - for the most part - arguments to that function! This technically makes Idem self documenting. But some arguments are global to all state definitions, such as the name, requisites, and order options.

7.1.4 Name and Names

Every state can take on a name argument, the name argument is always the primary identifier for a state. If the name argument is not provided, Idem will use the ID Declaration as the name.

The *names* argument allows for state replication to easily take place for multiple components. Using *names* can make it easy to define multiple identical resources in a clean way. Just as the *names* option and pass a list of desired names. Then Idem will compile the names down to multiple identical enforcements.

```
Some_machines:
  cloud.instance.present:
    - names:
      - web1
      - web2
      - web3
      - web4
      - web5
      - web6
      - web7
      - web8
      - web9
```

This state will create 9 identical cloud instances named web1 through web9

7.1.5 Order

The order keyword can change the evaluation order of Idem. When Idem runs, it evaluates the statements it is given in the order they are defined in SLS files. This means that, outside of requisites, Idem will run in the order it is defined.

The order of execution can be effectively nullified when using the parallel runtime. Idem can execute using either a serial, or a parallel runtime. The parallel runtime will evaluate all requisites and then run everything that is not encumbered by a requisite at the same time. This means that if you are using the parallel runtime, the order keyword will have no effect.

If you are using the serial runtime, then each state is executed one after another. This means that the order keyword can be used to change the order in which things are executed. Use the order keyword and pass in a number. The default ordering defined by Idem will add numbers based on the highest order value passed in. This means that if you pass *order: 1* then that state will be evaluated first. Similarly, you can pass *order: -1* and start with negative numbers to ensure that states are executed LAST.

```
State_A:
  cloud.instance.present:
    - order: -1  # Make it last
State_B:
  cloud.instance.present:
    - order: 1  # Make it first
```

Again, remember, that if you are using the parallel runtime, then both of these instances would be created at the same time.

7.2 Requisites

The requisite system inside of Idem is very powerful at determining the relationships that states have with each other. Being able to define requisites can make your enforcement significantly faster, more reliable, and can be used to create tasks.

These relationships are evaluated at runtime and can handle dynamic situations within your state definitions. A requisite is passed to a state as an argument:

```
State_A:
  cloud.instance.present:
    - require:
      - cloud.instance: State_B

State_B:
  cloud.network.present:
```

Is this case, State_A will only run once State_B has completed successfully. The referencing works by taking the *state ref* component of the *path ref* followed by the name or ID of the desired state to create a relationship with.

7.2.1 Requisite Ins

The requisites come in two flavors - *requisites*, and *requisite ins*. Every requisite that exists can be appended with an *_in* to specify that the direction changes.

A standard requisite states "I require you". For instance, this state is a standard requisite:

```
State_A:
  cloud.instance.present:
    - require:
      - cloud.network: State_B

State_B:
  cloud.network.present:
```

State_A is saying "I require State_B"

A requisite in simply says "They require me". This means that we can get the same effect as the requisite code above with this requisite in:

```
State_A:
  cloud.instance.present
```

(continues on next page)

(continued from previous page)

```
State_B:
  cloud.network.present:
    - require_in:
      - cloud.instance: State_A
```

So in this case, we are saying "I am State_B, State_A needs to require me".

7.3 Top Level Keys

Idem has a number of top level keys that can be used to include additional SLS files or to exclude specific IDs. You can also modify ID declarations from another file with the extend keyword.

7.3.1 Include

Include simply allow you to include information from another sls file in your run. Using include is simple, at the top of your SLS file just add *include* followed by a list of SLS references you wish to include in your run:

```
include:
  - foo.bar
  - azure.networks
```

The include statement evaluates SLS paths. You can easily execute Idem against a single SLS file, but Idem supports having a file tree. References to locations on the file tree are dot delimited and reference directories.

For instance, the SLS file you execute is assumed to be at the root of the tree. So if you execute an SLS file called *start.sls*, and it has the include statement:

```
include:
  - aws.instances
  - azure.networks
```

Then Idem will look for these files in a few locations. Idem will check the aws directory for a file called instances.sls, and if it does not find that file, it will check for a directory called *aws/instances/init.sls*.

Note that an SLS file can have *include* block along with states. Here is a possible aws/instances.sls:

```
include:
  - .ec2.vpcs

aws.resource-1:
    aws.resource-1.present:
        ....
```

An SLS file *aws/ec2/vpcs.sls* or *aws/ec2/vpcs/init.sls* is expected.

7.3.2 Extend

The extend keyword allows for diving into the state compiler and modifying a state from another included SLS file. This allows you to modify an external state. This can be useful if you are using your Idem code to manage multiple clouds that are NEARLY identical. So you can include the SLS files that define some external ID Declarations, then overwrite the options passed to them:

```
include:
  - gcp.networks

extend:
  Network_1:
    gcp.network.present:
      - ip_range: 10.10.57
```

In this case, the Network_1 ID from the gcp.networks SLS file will be overwritten with a new option.

RULES TO EXTEND BY

There are a few rules to remember when extending states:

1. Always include the SLS being extended with an include declaration
2. Requisites (watch and require) are appended to, everything else is overwritten
3. extend is a top level declaration, like an ID declaration, cannot be declared twice in a single SLS
4. Many IDs can be extended under the extend declaration

CHAPTER

EIGHT

SLS PARAMETERS

Writing and applying infrastructure as data relies on SLS files, where declarative data may be reapplied again and again to produce and maintain a desired result. In SLS files, parameters are how you describe and customize for the result that you want.

8.1 Creating a parameter file

A parameter file is a special SLS file that only contains key-value pairs as shown in the following example. A parameter file doesn't include any state declarations.

```
location: eastus
subscription_id: xxx-xxxxxxxxxxxx
locations:
    - eastus
    - westus
```

Parameter files can call other parameter files by using an `include` statement as shown in the following example.

```
subscription_id: xxx-xxxxxxxxxxxx
include:
    - params_extra
```

In the example, `params_extra.sls` then contains the following content.

```
locations:
    - eastus
    - westus
```

8.2 Calling parameters from a state

In SLS files with state declarations, parameters are available as a Python dictionary object called `params`.

Because `params` is a dictionary object, you can use dictionary functions such as get, items, and so on.

For example, you retrieve values with `params.get('parameter')` where `parameter` is the parameter name. Parameter values from the earlier example would be retrieved as shown here.

```
{{ params.get['location'] }}
{{ params['subscription_id'] }}
{{ params.get['locations'][1] }}
```

8.3 Default parameter values

To enforce a default value for the parameter, use `params.get('param', 'default_value')` where `default_value` is the value you want.

8.4 Missing parameter values

If a called parameter is missing from the parameters file or has no value `params.get('parameter')` returns `None` as the result.

To verify that a parameter is defined in the parameters file, use `params['missing_parameter']` where `missing_parameter` is the one you're looking for. If the parameter isn't defined, an exception similar to the following occurs.

```
Jinja variable:  'idem.idem.idem.idem.state. object' has no attribute 'missing_parameter'
```

8.5 Running an SLS state file and parameter file

In addition to defining parameters and referencing them within state files, commands need to specify the parameters to use.

To run a state file along with an associated parameter file, add the `--params` command line option.

```
idem state my_state.sls --params path/to/parameter_file.sls
```

8.6 Running an SLS state file and multiple parameter sources

Multiple parameter sources are supported. Locations specified in `--params` reference locations in `--param-sources` where each source is searched in the order given.

```
idem state my_state.sls --params "file.sls" "vault/location/specific" --param-sources
↪"file://local/file.sls" "vault://vault/location"
```

In the preceding command, `file.sls` is successfully found in the first parameter source `file://local/file.sls`.

Next, Idem checks for `vault/location/specific` in the first parameter source `file://local/file.sls`.

It isn't there, so Idem then checks for `vault/location/specific` in the second parameter source `vault://vault/location`.

All found sources are read and compiled into a single parameter tree.

8.7 Parameter precedence

Parameters can be overridden according to the following rules.

- A parameter value directly in a parameter file overrides a value coming from an included parameter file.
- In a parameter file with multiple included files, a value from a later included file overrides a value from an earlier one.
- In a command line that calls multiple parameter files, a parameter file from later in the command line overrides one given earlier.

See the following example, where `param.sls` is the parameter file called by the command line.

In this hierarchy of included parameters, *a* will be set to *4* and *b* will be set to *4*:

```
==> param.sls <==
include:
- param2

==> param2.sls <==
include:
- param3
- param4

==> param3.sls <==
a: 3
b: 3

==> param4.sls <==
a: 4
b: 4
```

If you change the example so that `param2.sls` reverses the include order, parameter *a* will be *3* and parameter *b* will be *3*.

```
==> param2.sls <==
include:
- param4
- param3
```

If you change the example so that `param2.sls` has its own assignment of *2* for parameter *a*, *a* will be *2* and *b* will be *3*.

```
==> param2.sls <==
include:
- param4
- param3
a: 2
```

CHAPTER

NINE

SLS PARAMETER VALIDATION

Parameter validation is feature essentially enables documentation and validation of params used in an SLS file. It is not at all related to *params* processing, as *params* processing only occurs during `state` sub-command execution - where actual param values are available. During validate phase we have no idea what params the given SLS uses. In fact, to extract those out of SLS is our goal.

9.1 Goal

The goal of SLS parameter validation is to extract/document parameters being used in an SLS file (including any files referred using `include` statement) for each state defined in the SLS file. Further, an additional goal is to do this transparently without exposing the end-user to any of the idem internals.

9.2 Limitation

When jinja processes any document it does not have any context as to what is the state that is being currently processed, since jinja is indifferent to sls syntax and only focuses on the piece of code it needs to handle. Not only that, the params may be getting used outside of any state, for initializing a jinja variables, like so:

```
{% set value = params.get('value') %}

state A:
    state.a.present:
        group: {{ value }}

state B:
    state.b.present:
        group: {{ value }}
```

The above limitations are the overbearing force behind this implementation.

9.3 Overview of the process involved

9.3.1 Step 1:- Transformation

We let jinja process the document, but instead of sending a traditional *dict* object as *params* object, we send an object of type **Parameters** class as defined in `idem/tool/parameter.py`. This class transforms orginal string (e.g. `{{ params.get('rg_name').get('your_rg', 'default') }}`) into another string which preserves the context (e.g. `?? params.get('rg_name').get('your_rg', 'default') ?? ^^rg_name^^.~~your_rg ??`). Here first portion of **??** string has the original expression and second portion helps us identify the state inside which the param is referred.

The above example transforms as:

```
state A:
state.a.present:
    group: ?? params.get('value') ?? ^^value^^ ??

state B:
state.b.present:
    group: ?? params.get('value') ?? ^^value^^ ??
```

As you can see above, the original parameter context, as well as the original `params.get()` string, are well preserved in the transformed YAML.

9.3.2 Step 2:- Extraction of parameters

This step involves extracting the parameters out of transformed YAML using relevant regular expressions. I have tried to capture the details within the file `idem/idem/validate/0001_find_params.py` itself.

9.3.3 Step 3:- Tallying with meta section in SLS

This step is simply giving warnings if a parameter used in any given state doesn't have a corresponding definition in the meta section of the SLS. This is just for aiding the SLS writer so that he/she can add a meta section if it is missing. Refer: `idem/idem/validate/0010_validate_meta.py`

9.3.4 Step 4:- Remapping transformed strings original values

This is simply mapping back transformed strings (e.g. `?? params.get('rg_name').get('your_rg', 'default') ?? ^^rg_name^^.~~your_rg ??`) to original string (e.g. `{{ params.get('rg_name').get('your_rg', 'default') }}`). Refer: `idem/idem/validate/0020_reverse_map.py`

9.4 Sample Output

For your reference, here is the output of *validate* sub-command on the above SLS:


```
{
    "high": {
        "state A": {
            "state.a.present": {
                "group": "{{ params.get('value') }}"
            },
            "__sls__": "test"
        },
        "state B": {
            "state.b.present": {
                "group": "{{ params.get('value') }}"
            },
            "__sls__": "test"
        }
    },
    "low": [
        {
            "state": "state.a.present",
            "name": "state A",
            "__sls__": "test",
            "__id__": "state A",
            "fun": "group",
            "order": 1
        },
        {
            "state": "state.b.present",
            "name": "state B",
            "__sls__": "test",
            "__id__": "state B",
            "fun": "group",
            "order": 1
        }
    ],
    "meta": {
        "SLS": {},
        "ID_DECS": {}
    },
    "parameters": {
        "GLOBAL": {
            "value": ""
        },
        "ID_DECS": {
            "test.state A": {
                "value": ""
            },
            "test.state B": {
                "value": ""
            }
        }
```


(continues on next page)

(continued from previous page)

```
    },
    "warnings": {
        "GLOBAL": [],
        "ID_DECS": {
            "test.state A": {
                "params_meta_missing": [
                    "value"
                ]
            },
            "test.state B": {
                "params_meta_missing": [
                    "value"
                ]
            }
        }
    }
}
```

You can see above two new sections are added in *validate* sub-command output, viz. *parameters* and *warnings*.

9.5 Some Additional Samples

Some examples with SLS and corresponding validation output.

1. **Iterating over list items**:

```
{% for ruleId in params.get('ruleIds') %}
{{ ruleId }}:
securestate.rules_status.present:
    - abc: def
{% endfor %}
```

```
{
    "parameters": {
        "GLOBAL": {
            "ruleIds": ""
        },
        "ID_DECS": {
            "resource_group.{{ params.get('ruleIds') }}": {
                "ruleIds": ""
            }
        }
    }
}
```

2. **Iterating over dict items**:

```
{% set ruleIds = params.get("ruleDict") %}
{% for key, value in ruleIds.items() %}
{{ key }}:
securestate.rules_status.present:
```

(continues on next page)

(continued from previous page)

```
    - x: value-{{ value }}
{% endfor %}
```

```
{
    "parameters": {
        "GLOBAL": {
            "ruleDict": ""
        },
        "ID_DECS": {
            "resource_group.key-{{ params.get('ruleDict') }}": {
                "ruleDict": ""
            }
        }
    }
}
```

3. **General use case of calling a function**:

```
{% set ruleIds = params.get("ruleIdsString") %}
{% for ruleId in ruleIds.split(',') %}
{{ ruleId }}:
securestate.rules_status.present:
    - x: y
{% endfor %}
```

```
{
    "parameters": {
        "GLOBAL": {
            "ruleIds": ""
        },
        "ID_DECS": {
            "resource_group.{{ params.get('ruleIdsString') }}": {
                "ruleIdsString": ""
            }
        }
    }
}
```

CHAPTER

TEN

ARGUMENT BINDING REFERENCES

An argument binding reference sets the state definition argument value to the result of another state execution. In this way, argument binding references determine the order of state execution in the structured layer state (SLS) file structure.

An argument binding reference uses the following format:

> "${<cloud>:<state>:<property_path>}"

Where <cloud> is the state cloud path reference (excluding function reference), <state> is the state declaration ID, and <property_path> is a colon (:) delimited path to the property value.

In the following example, State_B will be executed before State_A because the State_A argument "state_B_id" requires the "ID" value from State_B output.

```
State_A:
   cloud.instance.present:
     - name: "Instance A"
     - state_B_id: "${cloud:State_B:ID}"
 State_B:
   cloud.instance.present:
     - name: "Instance B"
```

10.1 Indexes

An argument binding reference can contain an index to point to a specific element of a collection property, as shown in the following example.

```
State_A:
  cloud.instance.present:
    - name: "Instance A"
    - state_B_address: "${cloud:State_B:nics[0]:address}"
State_B:
  cloud.instance.present:
    - name: "Instance B"
    - nics:
        - network_name: "Network_1"
          # address is populated after state is executed
          address:
        - network_name: "Network_2"
          # address is populated after state is executed
          address:
```

An argument binding reference can contain a wildcard (*) index to collect all elements in a collection property. In the following example, State_A "state_B_addresses" argument will be set to a list of 2 addresses, one address for each nic of State_B.

```
State_A:
  cloud.instance.present:
    - name: "Instance A"
    - state_B_addresses: "${cloud:State_B:nics[*]:address}"
State_B:
  cloud.instance.present:
    - name: "Instance B"
    - nics:
        - network_name: "Network_1"
          # address is populated after state is executed
          address:
        - network_name: "Network_2"
          # address is populated after state is executed
          address:
```

10.2 "Resource" Contract

To support argument binding, a cloud plugin must implement a "resource" contract, where every state execution function must return a "new_state" property as part of the return dictionary. The "new_state" is used to resolve argument binding requisites.

10.3 Arg_bind Requisites

Behind-the-scenes argument binding references are implemented using the Idem requisite system, where argument binding references are parsed during the SLS compilation phase and added to high data as arg_bind requisites. During arg_bind requisite execution, the "new_state" property returned after function execution is used to resolve the value of the referenced parameter.

The following example demonstrates SLS high data after the compilation phase, where "${cloud:State_B:ID}" is resolved as the arg_bind requisite.

```
State_A:
   cloud.instance.present:
     - name: "Instance A"
     - state_B_id: "${cloud:State_B:ID}"
     - arg_bind:
       - cloud:
          - State_B
             - ID: state_B_id
 State_B:
   cloud.instance.present:
     - name: "Instance B"
```

CHAPTER

ELEVEN

SLS INVERSION

In SLS function refs *present* and *absent* are complimentary to each other, and are used to ensure that a resource (corresponding to a state) gets created or gets deleted. Sometimes it is desirable to not write two different SLS files for just creating and delting some states. This is where SLS inversion tries to help. Command line argument `--invert` can be used to invert the behaviour of SLS file. However, this is not without limitations.

11.1 Motivation

The goal of SLS inversion is to use same SLS file to both create and delte resources as the case may be. All this can be understood with help of an example:

```
Assure Resource Group Present test_group:
  azure.resource_management.resource_groups.present:
  - resource_group_name: test_group
  - parameters:
      location: eastus
```

In the above SLS, we are creating a resource group. In the usual case to delete the above resource group we need to create a SLS file like so:

```
Assure Resource Group absent test_group:
  azure.resource_management.resource_groups.absent:
  - resource_group_name: test_group
  - parameters:
      location: eastus
```

With help of command-line parameter `--invert` we can create and delte the resource group using the same SLS, like so:

```
$ tail rg_create.sls
Assure Resource Group Present test_group:
azure.resource_management.resource_groups.present:
- resource_group_name: test_group
- parameters:
    location: eastus
$ idem state --output json rg_create.sls
{
    "azure.resource_management.resource_groups_|-Assure Resource Group Present test_
↪group_|-Assure Resource Group Present test_group_|-present": {
        "changes": {
```

(continues on next page)

(continued from previous page)


```
            "new": {
                "id": "/subscriptions/some-subscription/resourceGroups/test_group",
                "name": "test_group",
                "type": "Microsoft.Resources/resourceGroups",
                "location": "eastus",
                "properties": {
                    "provisioningState": "Succeeded"
                }
            }
        },
        "comment": "Created",
        "name": "Assure Resource Group Present test_group",
        "result": true,
        "old_state": null,
        "new_state": null,
        "__run_num": 1
    }
}
$ idem state --output json --invert rg_create.sls
{
    "azure.resource_management.resource_groups_|-Assure Resource Group Present test_
↪group_|-Assure Resource Group Present test_group_|-absent": {
        "changes": {
            "old": {
                "id": "/subscriptions/some-subscription/resourceGroups/test_group",
                "name": "test_group",
                "type": "Microsoft.Resources/resourceGroups",
                "location": "eastus",
                "properties": {
                    "provisioningState": "Succeeded"
                }
            }
        },
        "comment": "Accepted",
        "name": "Assure Resource Group Present test_group",
        "result": true,
        "old_state": null,
        "new_state": null,
        "__run_num": 1
    }
}
```


In this manner we can reverse changes done by an existing SLS file without actually writing a seperate SLS file.

11.2 State Requisite Handling

With SLS inversion, all state requisites also get inverted, in a sense that the order of execution of states is reversed. The idea behind this approach is to execute states in an inverted SLS in the reverse order of normal SLS. For example consider the following SLS:

```
sleep_mid:
  time.sleep:
  - require:
    - time: sleep_first
  - duration: 1

sleep_end:
  time.sleep:
  - require:
    - time: sleep_mid
  - duration: 1

sleep_independent:
  time.sleep:
  - duration: 1

sleep_first:
  time.sleep:
  - duration: 1
```

11.2.1 Normal Run

In a normal run (without `--invert`) the order of execution will be

1. sleep_first, sleep_independent
2. sleep_mid
3. sleep_end

```
$ idem state --output json invert.sls
{
    "time_|-sleep_independent_|-sleep_independent_|-sleep": {
        "comment": [
            "Successfully slept for 1 seconds."
        ],
        "old_state": {},
        "new_state": {},
        "name": "sleep_independent",
        "result": true,
        "__run_num": 1
    },
    "time_|-sleep_first_|-sleep_first_|-sleep": {
        "comment": [
            "Successfully slept for 1 seconds."
        ],
        "old_state": {},
```

(continues on next page)

(continued from previous page)

```
        "new_state": {},
        "name": "sleep_first",
        "result": true,
        "__run_num": 2
    },
    "time_|-sleep_mid_|-sleep_mid_|-sleep": {
        "comment": [
            "Successfully slept for 1 seconds."
        ],
        "old_state": {},
        "new_state": {},
        "name": "sleep_mid",
        "result": true,
        "__run_num": 3
    },
    "time_|-sleep_end_|-sleep_end_|-sleep": {
        "comment": [
            "Successfully slept for 1 seconds."
        ],
        "old_state": {},
        "new_state": {},
        "name": "sleep_end",
        "result": true,
        "__run_num": 4
    }
}
```

11.2.2 Inverted Run

With a `--invert` command-line parameter the order of state execution will be:

1. sleep_end, sleep_independent
2. sleep_mid
3. sleep_first

```
$ idem state --output json --invert invert.sls
{
    "time_|-sleep_end_|-sleep_end_|-sleep": {
        "comment": [
            "Successfully slept for 1 seconds."
        ],
        "old_state": {},
        "new_state": {},
        "name": "sleep_end",
        "result": true,
        "__run_num": 1
    },
    "time_|-sleep_independent_|-sleep_independent_|-sleep": {
        "comment": [
            "Successfully slept for 1 seconds."
```

(continues on next page)

(continued from previous page)

```
        ],
        "old_state": {},
        "new_state": {},
        "name": "sleep_independent",
        "result": true,
        "__run_num": 2
    },
    "time_|-sleep_mid_|-sleep_mid_|-sleep": {
        "comment": [
            "Successfully slept for 1 seconds."
        ],
        "old_state": {},
        "new_state": {},
        "name": "sleep_mid",
        "result": true,
        "__run_num": 3
    },
    "time_|-sleep_first_|-sleep_first_|-sleep": {
        "comment": [
            "Successfully slept for 1 seconds."
        ],
        "old_state": {},
        "new_state": {},
        "name": "sleep_first",
        "result": true,
        "__run_num": 4
    }
}
```

11.3 Requirement

To make SLS inversion work, all manadatory parameters required for *absent* and *present* for any given state should be present in the SLS, irrespective of actual function ref you are using. For example, the SLS file

```
Delete {{subnet}}:
  aws.ec2.subnet.absent:
  - name: {{VpcName}}
```

will not work with `--invert` command-line parameter. Since some mandatory parameters required by *present* are not provided. If the parameters required by *present* are also provided like below, inversion will work as expected with or without command-line parameter `--invert`.

```
Delete {{subnet}}:
  aws.ec2.subnet.absent:
  - name: {{VpcName}}
  - vpc_id: {{VpcId}}
  - cidr_block: 10.0.0.0/24
  - availability_zone: us-east-1d
  - tags:
    - Key: Name
      Value: one1
```

11.4 Limitations

While for some use cases `--invert` work well. It is not without limitations.

11.4.1 Argumnet binding does not work

Since *argument binding* involves uses output of one state to define input of another state, it doesn't work with SLS inversion.

CHAPTER

TWELVE

JMESPATH

`idem describe` is able to filter it's results using a tool called JMESpath. JMESpath is a query language for json.

When the `--filter` option is used with `idem describe`, the sls data gets changed into a format that is easy to use with jmespath.

For example, a traditional sls state in json format looks like this:

```
{
    "Description of test.succeed_with_comment": {
        "test.succeed_with_comment": [
            {"name": "succeed_with_comment"},
            {"comment": None},
        ]
    },
}
```

When performing a JMESpath search on the data, it first gets transformed to look like this:

```
[
    {
        "name": "Description of test.succeed_with_comment",
        "ref": "test.succeed_with_comment",
        "resource": [{"name": "succeed_with_comment"}, {"comment": None}],
    },
]
```

The data has been flattened into an list of dictionaries and the keys "name", "ref", and "resource" have been added for easy filtering. Don't worry, the end result is turned back into the sls form unless you supply the *–output=jmespath* flag.

You can always run `idem describe --output=jmespath` without `--filter` to see what the internal jmespath structure looks like.

12.1 Practicing with Static Data

Gathering data from the cloud can take a long time. When you are learning how to write JMESpaths, try writing a small script like this one to practice on static data:

```
# my_filter.py
import jmespath
import pprint
import sys
```

(continues on next page)

(continued from previous page)

```
# In this example, "data" is the output of "idem describe test --output=jmespath"
data = [
    {
        "name": "Description of test.anop",
        "resource": [{"name": "anop"}],
        "ref": "test.anop",
    },
    {
        "name": "Description of test.configurable_test_state",
        "resource": [
            {"name": "configurable_test_state"},
            {"changes": True},
            {"result": True},
            {"comment": ""},
        ],
        "ref": "test.configurable_test_state",
    },
    {"name": "Description of test.describe", "resource": [], "ref": "test.describe"},
    {
        "name": "Description of test.fail_with_changes",
        "resource": [{"name": "fail_with_changes"}],
        "ref": "test.fail_with_changes",
    },
    {
        "name": "Description of test.fail_without_changes",
        "resource": [{"name": "fail_without_changes"}],
        "ref": "test.fail_without_changes",
    },
    {
        "name": "Description of test.mod_watch",
        "resource": [{"name": "mod_watch"}],
        "ref": "test.mod_watch",
    },
    {
        "name": "Description of test.none_without_changes",
        "resource": [{"name": "none_without_changes"}],
        "ref": "test.none_without_changes",
    },
    {
        "name": "Description of test.nop",
        "resource": [{"name": "nop"}],
        "ref": "test.nop",
    },
    {
        "name": "Description of test.succeed_with_changes",
        "resource": [{"name": "succeed_with_changes"}],
        "ref": "test.succeed_with_changes",
    },
    {
        "name": "Description of test.succeed_with_comment",
        "resource": [{"name": "succeed_with_comment"}, {"comment": None}],
```

(continues on next page)

(continued from previous page)

```
        "ref": "test.succeed_with_comment",
    },
    {
        "name": "Description of test.succeed_without_changes",
        "resource": [{"name": "succeed_without_changes"}],
        "ref": "test.succeed_without_changes",
    },
    {
        "name": "Description of test.treq",
        "resource": [{"name": "treq"}],
        "ref": "test.treq",
    },
    {
        "name": "Description of test.update_low",
        "resource": [{"name": "update_low"}],
        "ref": "test.update_low",
    },
]

search_path = sys.argv[1]
pprint.pprint(jmespath.search(search_path, data))
```

12.2 Examples

Now for some examples of filtering with JMESpath. I will use the format of `idem describe test --filter="<JMESpath>"` in the following examples. If you called the little script we wrote above `my_filter.py` then the following two commands are equivalent. Keep that in mind as you move your one-off experiments to *idem describe*:

```
# Equivalent commands
my_filter.py "<JMESpath>"
idem describe test --output=pretty --filter="<JMESpath>"
```

Return only the states that use `test.update_low`

```
idem describe test --filter="[?ref=='test.update_low']"
```

output: .. code-block:: yaml

Description of test.update_low:
test.update_low: - name: update_low

Return only the states that start with "test.succeed"

```
idem describe test --filter="[?starts_with(ref, 'test.succeed']"
```

output: .. code-block:: yaml

Description of test.succeed_with_changes:
test.succeed_with_changes: - name: succeed_with_changes

Description of test.succeed_with_comment:
test.succeed_with_comment: - name: succeed_with_comment - comment: null

Description of test.succeed_without_changes:
test.succeed_without_changes: - name: succeed_without_changes

Return only tests that have "changes" in the state name:

```
idem describe test --filter="[?contains(name, 'changes')]"
```

output:

```
Description of test.fail_with_changes:
  test.fail_with_changes:
  - name: fail_with_changes
Description of test.fail_without_changes:
  test.fail_without_changes:
  - name: fail_without_changes
Description of test.none_without_changes:
  test.none_without_changes:
  - name: none_without_changes
Description of test.succeed_with_changes:
  test.succeed_with_changes:
  - name: succeed_with_changes
Description of test.succeed_without_changes:
  test.succeed_without_changes:
  - name: succeed_without_changes
```

Return only states that have "succeed_with_comment" in the "name" parameter

```
idem describe test --filter="[?resource[?name=='succeed_with_comment']]"
```

output:

```
Description of test.succeed_with_comment:
  test.succeed_with_comment:
  - name: succeed_with_comment
  - comment: null
```

12.3 Learn More

https://jmespath.org/tutorial.html

https://jmespath.org/examples.html

https://jmespath.org/specification.html

https://pypi.org/project/jmespath

https://docs.aws.amazon.com/sdk-for-php/v3/developer-guide/guide_jmespath.html

https://docs.microsoft.com/en-us/cli/azure/query-azure-cli

https://www.azurecitadel.com/cli/jmespath/

CHAPTER

THIRTEEN

TRANSPARENT REQUISITES

Transparent requisites is a powerful feature inside of Idem. It allows requisites to be defines on a function by function basis. This means that a given function can always requires any instance of another function, in the background. This makes it easy for state authors to ensure that executions are always executed in the correct order without the end user needing to define those orders.

It is easy to do, at the top of your system module just define the *TREQ* dict, this dict defines what functions will require what other functions:

```
TREQ = {
    "treq": {
        "require": [
            "test.nop",
        ]
    },
}
```

This stanza will look for the function named *treq* inside of the module that it is defined in, then it will add *require : - test.nop* for every instance found of *test.nop* in the current run. If *test.nop* is never used, then no requisites are set. Any requisite can be used, and multiple requisites can be used.

13.1 Unique Transparent Requisite

Another type of transparent requisite is *unique*. A function can be declared *unique* to prevent concurrent executions. The *unique* transparent requisite is significant in case of a parallel execution (default). *TREQ* dict at the top of the system module define *unique* with a list of functions within the module.

```
TREQ = {
    "unique": [
        "test.create",
        "test.delete",
    ]
}
```

In the example above, the instances of *test.create* within the current run will be invoked serially, and all the instances of *test.delete* will be invoked serially. Instance of *test.create* and *test.delete* can be invoked in parallel. It is achieved by selecting a single instance of the unique function, and setting the other instances of the same function as dependent on it. During the next run, a new instance will be selected. The *unique* requisite is re-evaluated in each run.

CHAPTER

FOURTEEN

SECURE MULTIPLE ACCOUNT MANAGEMENT

You can run Idem against multiple cloud accounts and providers. The Idem *acct* tool lets you specify cloud account and provider information in a file. The *acct* tool is a dependency of Idem. It is used to encrypt the file that stores the account information securely on the file system.

Support for file-based authentication was added as of Idem 6. Additional authorization mechanisms are expected in future Idem releases.

14.1 Static Account Management

In this example, you create a file in which to store credentials. The file is a simple YAML file that can store credentials for multiple providers and accounts.

The following example *creds.yml* file includes sample aws system values. The only profile shown is an aws default profile, but you could have multiple sections with profiles for more providers and accounts.

```
aws:
  default:
    aws_access_key_id:
    aws_secret_access_key:
    region_name:
```

After creating the file with credentials in it, run the *acct* tool to encrypt the file:

```
$ acct encrypt creds.yml
New encrypted file created at: creds.yml.fernet
The file was encrypted with this key:
j-ytfz45n2wRUHDZJsumtG5_Dih3b3lTA1P2apqNuFg=
```

Now you have an encrypted credentials file and a key to access it. Keep the key in a safe place.

To run *idem* with credentials stored in the file, use the *–acct-file* and *–acct-key* options.

In addition, you can use the *–acct-profile* option to select a profile from within a credentials file that contains multiple profiles. In the example above, *default* is the account profile.

If there are multiple profiles, and you don't supply the *–acct-profile* option, the *default* profile is used.

If you don't want to pass account information as CLI options, you can set the following environment variables:

```
export ACCT_FILE=<full path to creds.yml.fernet>
export ACCT_KEY=<creds file encryption key>
```

14.2 ACCT RENDER PIPES

Before an acct_file is encrypted, it will be passed through the specified acct render pipes. The default render pipe is "jinja|yaml"

```
$ idem encrypt credentials.yaml --render-pipe="jinja|yaml"
```

14.3 UNENCRYPTED ACCT FILE

If no ACCT_KEY is provided, then acct will assume that the ACCT_FILE is unencrypted.

For states/exec modules to specify a custom acct render pipe, it needs to be specified in the idem config file.

```
# idem-config.cfg
acct:
  render_pipe: jinja|yaml
```

14.4 ALLOWED_BACKEND_PROFILES

If the idem config file specifies allowed_backend_profiles, then only backend profiles with names in this list will be processed by acct. The default is to process ALL acct backend profiles.

The following config file shows 3 profiles that are allow-listed in the idem config file:

```
# idem-config.cfg
acct:
  allowed_backend_profiles:
    - allowed_backend_profile_name_1
    - allowed_backend_profile_name_2
    - allowed_backend_profile_name_3
```

The following unencrypted credentials file has multiple profiles for account backends under the "vault" and "lastpass" providers. Each `acct-backend` profile contains other normal profiles for acct to use. For example, a vault acct-backend may connect to a vault data store that contains acct profiles for connecting to aws and azure. The vault acct-backend profile contains credentials for connecting to vault. The vault acct-backend plugin connects to vault and collects more credentials for idem projects from vault. Only profile names that match the "acct:allowed_backend_profiles" config option will be used to collect more credentials from the acct backend profiles. This way, a user can be selective about which acct-backend to use in the case of conflicts.

```
$ idem exec test.ping --acct-file=credentials.yaml --config=idem-config.cfg
```

14.5 ACCT SERIAL PLUGIN

The *pop-serial* plugin that is used by acct to serialize acct data before it is encrypted can be specified in the idem config file. The default plugin for serializing data in acct is "msgpack":

```
# idem-config.cfg
acct:
  serial_plugin: msgpack
```

CHAPTER

FIFTEEN

ACCT FILE

In Idem, you can supply credentials for many different environments. Credentials are stored in a single encrypted file. An account credentials file follows this pattern:

```
credential_provider_1:
    profile_1:
        key_1: value_1
        key_2: value_2
    profile_2:
        key_1: value_1
        key_2: value_2

credential_provider_2.acct_sub:
    profile_1:
        key_1: value_1
        key_2: value_2
    profile_2:
        key_1: value_1
        key_2: value_2
```

15.1 providers

In the account file, the first level of keys are the `provider` keys. For every system that uses `ACCT` for authentication, there's a top-level Python file that specifies the provider keys that are acceptable for authenticating to that system. The general format for the Python code follows this pattern:

```
def __init__(hub):
    hub.my_dyne.my_subsystem.ACCT = ["my_provider"]
```

The code above enables the "my_provider" provider keys to authenticate "my_dyne.my_subsystem". "my_dyne" could be "exec", "states", "tool", "evbus", "esm", "sources", or another dynamic namespace. "my_subsystem" is the root folder name of your cloud-specific code under the dynamic namespace.

15.2 acct plugins

Provider keys can specify an account plugin that performs additional processing for a profile. In the following example, the aws.gsuite account plugin uses a Google username and password to obtain valid tokens and keys for idem-aws.

```
aws.gsuite:
    my_profile:
       username: my_google_user
       password: my_google_password
```

15.3 profiles

The second level of keys in the account file are the `profiles` under each provider.

The default profile is usually named "default" if no other profile is named. The "default" name is only an optional convention, not a requirement. Some components, like *evbus*, don't use default profiles.

You can add multiple profiles under a provider, where each profile under the same provider has a unique name. Duplicate profile names must be under different providers. For example, a "default" AWS profile and "default" Azure profile are acceptable.

Profile names must match regex '[-.w]+'; for ASCII text, this includes a-z, A-Z, 0-9, _ . and -.

```
aws:
  default:
    id: XXXXXXXXXXXXXXXX
    key: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
    region: us-east-1
azure:
  default:
    client_id: "aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaaa"
    secret: "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
    subscription_id: "bbbbbbbb-bbbb-bbbb-bbbb-bbbbbbbbbbbb"
    tenant: "cccccccc-cccc-cccc-cccc-cccccccccccc"
```

Each system specifies and uses profiles differently. See the *evbus*, *state*, *sources*, and *esm* documentation for details on how each specifies and uses profiles.

15.4 backends

External credential stores can also contain account profile information. In the account file, these are specified under the "acct-backend" top-level key:

```
acct-backend:
  lastpass:
      username: user@example.com
      password: password
      designator: acct-provider
  keybase:
      username: user
      password: password
```

15.5 extras

Some plugins make use of non-secret values in their authentication methods. These are specified in the Idem config file under `acct.extras`.

```
acct:
  extras:
    my_provider:
      my_profile:
        my_key: my_non_secret_value
```

In code, you access extras via `hub.OPT` as shown in the following example:

```
def gather(hub, provider: str, profile: str) -> dict:
    return hub.OPT.acct.extras[provider][profile]
```

CHAPTER

SIXTEEN

IGNORE_CHANGES REQUISITE

ignore_changes can be used in sls blocks to prevented parameters to be updated on brown-field resources. If a parameter is specified under the ignore_changes and this parameter will be overridden with None and present() function will ignore updating such None-value parameters.

In the following example, State_A is a green-field resource. During the first Idem run, Idem will create State_A resource with the tags value. However, since ignore_changes contains "tags", during the second Idem run to update the State_A resource, Idem will not update tags even when the tags of the resource has been deviated away from the initial {tag-key: tag-value} value.

```
State_A:
   cloud.instance.present:
     - name: my-resource
     - tags: {tag-key: tag-value}
     - ignore_changes:
       - tags
```

Nested data under a parameter can be specified with a syntax similar to what arg_binding use: ':' for traversing a dict structure and '[]' for traversing a list. In addition, '[*]' can be used to traverse through all elements in a list. For example: Given the tags input below, if we want to just ignore changing tag1, we can do tags[0]. Though, we need to be careful on ignoring list type data with indexing here, since the order of list elements is not guaranteed in Idem.

```
tags:
  - Name: tag1
    Value: value1
  - Name: tag2
    Value: value2
```

If the parameter path under ignore_changes is invalid, Idem will output a warning message but it won't fail the resource management operation.

Note: ignore_changes requisite only takes into effect on a brown-field resource. That is, the enforced State_A exists in ESM cache or the resource_id has been supplied in sls file.

CHAPTER

SEVENTEEN

RECREATE_ON_UPDATE REQUISITE

When Idem can't update an existing resource, use `recreate_on_update` to delete the resource and recreate it.

In the following examples, State_A isn't supported for updates. The only way to update values in State_A is to create a new resource and delete the one that has the old values. `name_prefix` argument is used to create a unique name beginning with the specified prefix for State_A.

17.1 Greenfield Example 1

In the following greenfield example, State_B is dependent on State_A output. State_B parameter_2 requires the State_A resource_id.

Because this is a greenfield deployment, and State_A will be new, the `recreate_on_update` shown in State_A is never activated.

```
State_A:
  cloud.instance.present:
    - name_prefix: my-resource-A
    - parameter_1: value-1
    - parameter_2: value-2
    - ignore_changes:
      - parameter_2
    - recreate_on_update:
        create_before_destroy: true

State_B:
  cloud.instance.present:
    - name: Instance-B
    - parameter_1: value-1
    - parameter_2: "${cloud:State_A:resource_id}"
```

With this setup Idem generates a unique name for your State_A and can then update the State_B without conflict before destroying the previous State_A.

The console output for the example is:

```
ID: State_A
Function: cloud.instance.present
Result: True
Comment: ("Created cloud.instance 'my-resource-A-1'",)
Changes:
```

(continues on next page)

(continued from previous page)

```
new:
----------
    name:
        my-resource-A-1
    name_prefix:
        my-resource-A
    resource_id:
        my-resource-A-1
    parameter_1:
        value-1
    parameter_2:
        value-2

------
ID: State_B
Function: cloud.instance.present
Result: True
Comment: ("Created cloud.instance 'Instance-B'",)
Changes:
new:
----------
    name:
        Instance-B
    parameter_1:
        value-1
    parameter_2:
        my-resource-A-1
```

17.2 Brownfield Example 1

In the following brownfield example, State_A needs to update its parameter_1 value. Because this is a brownfield example, and State_A isn't supported for updates, it must be recreated with the new value.

Note that `create_before_destroy` is set to true so that Idem can create the new State_A resource, supply its resource_id to State_B, and delete the old State_A resource afterward. Without `create_before_destroy` being true, there might have been a gap during which State_B couldn't get the resource_id.

```
State_A:
  cloud.instance.present:
    - name_prefix: my-resource-A
    - resource_id: my-resource-A-1
    - parameter_1: value-1-updated
    - parameter_2: value-2
    - ignore_changes:
      - parameter_2
    - recreate_on_update:
        create_before_destroy: true

State_B:
  cloud.instance.present:
```

(continues on next page)

(continued from previous page)

```
    - name: Instance-B
    - parameter_1: value-1
    - parameter_2: "${cloud:State_A:resource_id}"
```

The console output for the example is:

```
ID: State_A
Function: cloud.instance.present
Result: True
Comment: ("Created cloud.instance 'my-resource-A-2'",)
Changes:
new:
----------
    name:
        my-resource-A-2
    name_prefix:
        my-resource-A
    resource_id:
        my-resource-A-2
    parameter_1:
        value-1-updated
    parameter_2:
        value-2

------
ID: State_B
Function: cloud.instance.present
Result: True
Comment: ("Updated cloud.instance 'Instance-B'",)
Changes:
old:
-----
    parameter_2:
        my-resource-A-1
new:
----------
    parameter_2:
        my-resource-A-2

------
ID: State_A_delete_old
Function: cloud.instance.present
Result: True
Comment: ("Deleted cloud.instance 'State_A_delete_old'",)
Changes:
old:
----------
    name:
        State_A_delete_old
    name_prefix:
        my-resource-A
    resource_id:
```

(continues on next page)

(continued from previous page)

```
        my-resource-A-1
    parameter_1:
        value-1
    parameter_2:
        value-2
```

17.3 Brownfield Example 2

In the following brownfield example, State_A needs to update its parameter_1 value. State_A isn't supported for updates, so it must be recreated with the new value.

In this case, State_A doesn't have any dependent resources, so `create_before_destroy` can be false. Idem can safely delete the old State_A resource before creating the new one.

```
State_A:
  cloud.instance.present:
    - name_prefix: my-resource-A
    - resource_id: my-resource-A-1
    - parameter_1: value-1-updated
    - parameter_2: value-2
    - ignore_changes:
      - parameter_2
    - recreate_on_update:
        create_before_destroy: false
```

The console output for the example is:

```
ID: State_A_delete_old
Function: cloud.instance.present
Result: True
Comment: ("Deleted cloud.instance 'State_A_delete_old'",)
Changes:
old:
----------
    name:
        State_A_delete_old
    name_prefix:
        my-resource-A
    resource_id:
        my-resource-A-1
    parameter_1:
        value-1
    parameter_2:
        value-2

-----------

ID: State_A_create_new
Function: cloud.instance.present
Result: True
Comment: ("Created cloud.instance 'my-resource-A-2'",)
```

(continues on next page)

(continued from previous page)

```
Changes:
new:
----------
    name:
        my-resource-A-2
    name_prefix:
        my-resource-A
    resource_id:
        my-resource-A-2
    parameter_1:
        value-1-updated
    parameter_2:
        value-2
```

17.4 Greenfield Example 2

In the following greenfield example, State_A will be newly created, including its tags. Remember, State_A is still unsupported for updates.

Because `ignore_changes` contains `tags`, if tag keys or values have drifted from their newly created states, a subsequent Idem run to bring the resource back into tag compliance won't recreate the resource, even though `recreate_on_update` is present.

Note that, in addition, the subsequent run won't bring the tag keys or values back into compliance.

```
State_A:
   cloud.instance.present:
     - name_prefix: my-resource
     - tags: {tag-key: tag-value}
     - ignore_changes:
       - tags
     - recreate_on_update:
         create_before_destroy: false
```

CHAPTER

EIGHTEEN

USING A DELAY BETWEEN STATES TO RESOLVE JINJA TEMPLATE ARGUMENT BINDING

Idem SLS files support a dependency delay between states, where a state isn't rendered until a preceding state has been rendered:

```
State_A:
  test.nop
#!require: State_A
State_B:
  test.nop
```

You can use a delay to parse and process an argument binding reference in a Jinja template. A Jinja template argument binding reference follows this pattern:

```
{% for key,value in hub.idem.arg_bind.resolve('${<cloud>:<state>}').items() %}
    {{ value }}
{% endfor %}
```

In the following example, State_B includes a Jinja template argument binding reference that needs a value from State_A. The *#!require:State_A* delay forces the rendering of State_B to first wait for State_A to be rendered, which makes its *Name* tag value available to State_B.

```
State_A:
  cloud.subnetwork.search:
    - tags: {Name: {% params.get("pvt_subnetwork_name") %}}

#!require:State_A

State_B:
  cloud.private_cloud_attachment.present:
    - name: "Private cloud B"
    - subnetwork_ids: {% subnetwork_ids=[] %} {% for k,v in  hub.idem.arg_bind.resolve('$
↪{cloud.subnetwork:State_A}').items() %} {{ subnetwork_ids.append(v["resource_id"]) }} {
↪% endfor %}
```

18.1 Fetching argument binding reference values

For a Jinja template containing an argument binding reference to be rendered, the argument binding must be passed to custom function *hub.idem.arg_bind.resolve()* as a string. The function parses the argument binding template and resolves the value.

The *new_state* of the prerequisite block executed states should be available in *hub.idem.RUNS[name]["running"]* using the argument binding references in Jinja that were resolved in the *rend* subsystem.

For List, data resolution happens as below - The target 'foo:bar:[0]' or 'foo:bar[0]' will return data['foo']['bar'][0] if data like {'foo':{'bar':['baz']}}

For Dict, data resolution happens as below - The target 'foo:bar:0' will return data['foo']['bar'][0] if data like {'foo':{'bar':{'0':'baz'}}}

CHAPTER

NINETEEN

DELAYED RENDERING

By default, the states in Idem SLS files are rendered, compiled, and executed all at once. If you need to separate them, Idem supports a dependency delay between states.

To render states separately, add the following line before the state that you need to delay.

```
#!require:<prerequisite_state>
```

In the following example, State_B isn't rendered until State_A has been rendered:

```
State_A:
  test.nop
#!require: State_A
State_B:
  test.nop
```

19.1 Closing a delayed state block

Expanding on the preceding example, if you have State_C that's safe to run in parallel with State_A, you can close the delayed State_B block:

```
State_A:
  test.nop
#!require: State_A
State_B:
  test.nop
#!END
State_C:
  test.nop
```

Otherwise, both State_B and State_C render after State_A.

If no *#!END* is provided, all blocks close at the end of the SLS file.

CHAPTER

TWENTY

SENSITIVE REQUISITE

Sensitive requisite can be used in sls blocks to prevented parameters to be outputted to console.

For Idem state resources that implement the "resource" contract, parameters specified under the sensitive requisite are hidden from the "changes" output.

In the following example, State_A uses sensitive requisite to hide its secret from "changes" output.

```
State_A:
  cloud.instance.present:
    - name: my-resource
    - public: public-data
    - secret: secret-data
    - sensitive:
      - secret
```

Assume present() does a creating operation, the console output of this Idem run will be:

```
changes:
  new:
    name: my-resource
    public: public-data
```

Note: sensitive requisite only hides data in "changes" that is outputted to console. All data will still be saved into ESM cache in plain text.

CHAPTER

TWENTYONE

SLS ACCT

Each state in an SLS file defaults to using the "default" profile for the associated provider.

Idem determines which profile is appropriate by looking for `ACCT` on the hub in the root of the state definition. For example, in `idem_aws/states/aws/init.py` we might see the following code:

```
def __init__(hub):
    hub.states.aws.ACCT = ["aws"]
```

The code specifies that all plugins under `hub.states.aws` should use profiles under the "aws" provider key in the account file.

An account file with AWS credentials might look like this:

```
aws:
  default:
    id: XXXXXXXXXXXXXXXX
    key: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
    region: us-east-1
  other:
    id: XXXXXXXXXXXXXXXX
    key: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
    region: us-east-1
```

A single SLS file can reference multiple AWS credential profiles as shown in the following example:

```
# No profile is specified, so the "default" AWS profile is used.
ensure_vpc:
  aws.vpc.present:
     - kwarg1: value1

# The "other" AWS profile is specified for use.
ensure_vpc:
  aws.vpc.present:
     - acct_profile: other
     - kwarg1: value1
```

States from one cloud can depend on states from another cloud. Idem will associate the right profiles with the right plugins and keep the profiles separate in their own contexts.

Consider the following multi-cloud account file:

```
aws:
  default:
```

(continues on next page)

(continued from previous page)

```
    id: XXXXXXXXXXXXXXXX
    key: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
    region: us-east-1
azure:
  default:
    client_id: "aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaaa"
    secret: "xxxxxxxxxxxxxxxxxxxxxxxxxxxx"
    subscription_id: "bbbbbbbb-bbbb-bbbb-bbbb-bbbbbbbbbbbb"
    tenant: "cccccccc-cccc-cccc-cccc-cccccccccccc"
```

Profiles under the "aws" provider are only sent to plugins that specify the "aws" provider for authenticaton. Profiles under the "azure" provider are only sent to plugins that specify the "azure" provider for authenticaton.

```
# The "default" "aws" profile is used.
ensure_vpc:
  aws.vpc.present:
     - kwarg1: value1

# The "default" "azure" profile is used.
ensure_vpc:
  azure.vpc.present:
     - kwarg1: value1
```

In this way, Idem seamlessly integrates multiple requisites and profiles across different clouds, all within a single SLS file.

21.1 Aggregate State

You can dynamically extend account information using the `acct.profile` state. In the following example, `test.present` represents data from an arbitrary resource. Its values are passed via `arg_bind` to an `acct.profile` state. The third state requires the `new_profile` state and then makes use of the profile created in that state.

Warning: Use this feature with arg_binding to use information generated in one state as credentials in other states. You can use the feature to create roles and dynamically assume those roles in subsequent states. Do NOT use this feature to bypass encrypting your credentials file. Exercise caution and do not abuse this feature.

```
mock_acct:
  test.present:
    - new_state:
        key_1: value_1
        key_2: value_2

new_profile:
  acct.profile:
    - provider_name: test
    - key_1: ${test:mock_acct:key_1}
    - key_2: ${test:mock_acct:key_2}

test_result:
```

(continues on next page)

(continued from previous page)

```
  test.acct:
    - acct_profile: new_profile
    - require:
      - acct: new_profile
```

Note: This can also be done without "require" blocks by using the *reconciliation loop*.

21.2 Single-use Profiles

In the following code block, acct_data is passed directly into the state. The profiles defined here will be used in place of all the profiles defined for the rest of the RUN. This acct_data is not preserved and only exists in the context of the state that uses it.

```
mock_acct:
  test.present:
    - new_state:
        key_1: value_1
        key_2: value_2

test_result:
  test.acct:
    - acct_profile: new_profile
    - acct_data:
        profiles:
          test:
            new_profile:
              key_1: ${test:mock_acct:key_1}
              key_2: ${test:mock_acct:key_2}
```

If arg_binding is not required, account data that isn't sensitive can be saved in a Jinja variable and explicitly passed to each state that needs it. However, you get better value and security from writing an acct plugin.

```
{% set acct_data = {"profiles": {"test": {"new_profile": {"key_1": "value_1", "key_2":
↪"value_2"}}}} %}

test_result:
  test.acct:
    - acct_profile: new_profile
    - acct_data: {{acct_data}}
```

21.3 Copy From Existing Profiles

To copy from an existing profile, specify the "source_profile" key in the `acct.profile` state. The profile matching the "source_profile" name under the given provider will be used as a base for constructing the new profile.

Consider the following acct_file:

```
test:
  source:
    key_1: overwritten
    key_3: copied
```

The following state copies the existing profile under the given provider in the acct_file. "key_1" is defined in both places, so the new profile will overwrite that value. "key_3" is not defined in the new_profile, so it will be copied from the existing profile in the acct_file to the new profile.

```
new_profile:
  acct.profile:
    - provider_name: test
    - source_profile: source
    - key_1: value_1
    - key_2: value_2

test_result:
  test.acct:
    - acct_profile: new_profile
    - require:
      - acct: new_profile
```

CHAPTER

TWENTYTWO

SLS SOURCES

SLS sources are directory trees, archives, and remote stores that contain sls files. SLS and param sources can come from many different places. The plugins that can be used to process SLS sources are in idem/idem/sls.

The format for an sls sources is:

```
<protocol>://<resource>
```

The format for authenticated sls sources is:

```
<protocol_plugin>://<acct_profile>@<resource>
```

The named acct profile associated with the protocol_plugin provider will have its values passed to `ctx.acct` of the appropriate "cache" function.

File sources that have a mimetype, such as zip files, will be unarchived before further processing.

This is an example of an idem config file that specifies `sls_sources` and `param_sources`:

```
idem:
  sls_sources:
    - file://path/to/sls_tree
    - file://path/to/sls_source.zip
    - git://github.com/my_user/my_project.git
    - git+http://github.com/my_user/my_project.git
    - git+https://github.com/my_user/my_project.git
  param_sources:
    - file://path/to/sls_tree
    - file://path/to/sls_source.zip
    - git://github.com/my_user/my_project.git
    - git+http://github.com/my_user/my_project.git
    - git+https://github.com/my_user/my_project.git
```

`sls_sources` and `param_sources` can also be specified from the CLI.

```
$ idem state my.sls.ref \
      --sls-sources \
      "file://path/to/sls_tree" \
      "file://path/to/sls_source.zip" \
      "git://github.com/my_user/my_project.git" \
      "git+http://github.com/my_user/my_project.git"
      "git+https://github.com/my_user/my_project.git"
      --param-sources \
      "file://path/to/sls_tree" \
```

(continues on next page)

(continued from previous page)

```
    "file://path/to/sls_source.zip" \
    "git://github.com/my_user/my_project.git"
    "git+http://github.com/my_user/my_project.git"
    "git+https://github.com/my_user/my_project.git"
```

CHAPTER

TWENTYTHREE

THE SLS TREE

SLS files can be arranged in a tree fo easy maintinence and development. This allows you to arrange your Idem code in a way that makes it easy to keep track of things, but also makes it easy to modify other SLS files, make code reuasable, and more!

When you run Idem and just give it a file to run, it assumes that your SLS tree is rooted in that file's directory. This makes it easy to make multiple files that can be called together while just starting from an easy to find entry point.

Lets assume that you run *idem state start.sls* in a directory called *idem/*. Then you have defined that the directory called *idem/* is the root of your SLS tree. Now all of the SLS references inside *include* directives are references relative to the *idem/* directory.

SLS tree paths are resolved in one of two ways. An SLS reference of *foo.bar* will resolve to either *foo/bar.sls* or to *foo/bar/init.sls*. This allows for code to be easily organized into directories that contain groupings of code.

CHAPTER

TWENTYFOUR

EXEC STATE

Exec modules can be run from SLS using the "exec.run" state. The return from the exec module is put in the state's "new_state", so it can be used in arg_binding. The first comment in the exec module state return is the cli command that can be used to call the exec module. The `path` is the reference path to the exec module. An exec module named "func" in the directory "project_root/project/exec/exec_sub/plugin.py" has a `path` of "exec.exec_sub.plugin.func". It can be accessed on the hub via "hub.exec.exec_sub.plugin.func()". The `path` that should be passed to the `exec.run` state would be "exec_sub.plugin.func" The acct_profile will be used to create a ctx based on the appropriate provider for the given `path`.

```
exec_func:
  exec.run:
    - path: test.more
    - acct_profile: default
    - args:
      - arg1
      - arg2
      - arg3
    - kwargs:
      kwarg_1: val_1
      kwarg_2: val_2
      kwarg_3: val_3
```

Output of running the state with `--output=json`:

```
{
"exec_|-exec_func_|-exec_func_|-run": {
    "tag": "exec_|-exec_func_|-exec_func_|-run",
    "name": "exec_func",
    "changes": {},
    "new_state": {
        "args": [
            "arg1",
            "arg2",
            "arg3"
        ],
        "kwargs": {
            "kwarg_1": "val_1",
            "kwarg_2": "val_2",
            "kwarg_3": "val_3"
        },
        "ctx": {
            "acct": {}
```

(continues on next page)

(continued from previous page)

```
        }
    },
    "old_state": {},
    "comment": [
        "idem exec test.more --acct-profile=default arg1 arg2 arg3 kwarg_1=val_1 kwarg_
↪2=val_2 kwarg_3=val_3"
    ],
    "result": true,
    "esm_tag": "exec_|-exec_func_|-exec_func_|-",
    "__run_num": 1,
    "start_time": "2022-05-31 15:08:13.548936",
    "total_seconds": 0.001457
    }
}
```

CHAPTER

TWENTYFIVE

SLS RESOLVER PLUGINS - HUB.SOURCE

Idem SLS resolvers are very easy to write and can often work in just a few lines of code. They are implemented in a nested pop subsystem under idem, this means that you can vertical app-merge these plugins into idem if that makes the most sense.

The vast majority of the work to gather SLS files is completely generic, so adding new sources should be very easy. For instance, the code to load from the filesystem is just a few lines of code.

First create a directory at my_project_root/my_provider/source. In your project's conf.py, extend idem's namespace with your "source" directory.

```
# my_project_root/my_provider/conf.py
DYNE = {"source": ["source"]}
```

Now create a plugin in your "source" directory.

The plugin simply needs to implement the *cache* async function. This function receives the *hub*, *protocol*, *source*, and *loc*. These will be passed into the function.

The *protocol* is usually just the name of the resolver plugin. Sometimes it will contain an additional protocol like "git+https"

The *source* is the root path to pull the file from. In the case of the file resolver this will be *file:///path/to/sls/root*.

The *location* or location is the desired file location relative to the *source*.

```
# my_project_root/my_provider/source/my_provider.py

# Generic python imports
from typing import ByteString
from typing import Tuple
import tempfile

try:
    # Import plugin-specific libraries here
    import my_provider.sdk

    HAS_LIBS = (True,)
except ImportError as e:
    HAS_LIBS = False, str(e)


def __virtual__(hub):
    # Perform other dependency checks as needed here I.E check for "git" or "fuse"␣
↪installed via the OS
```

(continues on next page)

(continued from previous page)

```
    return HAS_LIBS


# The virtualname is how the provider will appear on the hub,
# use this to avoid clashes with python module names and python keywords
__virtualname__ = "my_provider"


def __init__(hub):
    # This tells idem that the "my_provider" key in the acct file contains profiles for␣
↪this plugin
    hub.source.my_provider.ACCT = ["my_provider"]


async def cache(
    hub, ctx, protocol: str, source: str, location: str
) -> Tuple[str, ByteString]:
    """
    Read data from my_provider
    :param hub:
    :param ctx: ctx.acct will contain the appropriate credentials to connect to my sls␣
↪source's sdk
    :param source: The url/path/location of this sls source
    :param location: The path/key to access my SLS data relative to the source


    Define what credentials should be used in a profile for "my_provider"

    .. code-block:: sls

        my_provider:
          my_profile:
            my_kwarg_1: my_val_1
            other_arbitrary_kwarg: arbitrary_value
    """
    # ctx.acct will have the credentials for the profile specified in the sls-sources/
↪params-sources string
    data: bytes = my_provider.sdk.read(**ctx.acct)

    # There are two ways we can pass the sources on to idem
    if "option 1":
        # Store the data returned from the sdk in memory (preferred method)
        return f"{location}.sls", data
    elif "option 2":
        # Cache the data returned from the sdk in a temporary location
        with tempfile.NamedTemporaryFile(suffix=".sls", delete=True) as fh:
            fh.write(data)
            fh.flush()
            # Process the cached data like a traditional sls file source
            return await hub.source.file.cache(
                ctx=None, protocol="file", source=fh.name, location=location
            )
```

Your sls source can be invoked from the CLI like so:

```
$ idem state location --sls-sources "my_provider://my_profile@<source>" --params-sources
↪"my_provider://my_profile@<source>"
```

CHAPTER

TWENTYSIX

GROUP PLUGINS - HUB.GROUP

Group plugins are capable of organizing and manipulating the return data of an idem run before it is rendered.

26.1 Finding Group Plugins

Use `idem doc` to list all the available group plugins:

```
$ idem doc group | grep -o "^group.\w*"
```

You can also use `jq` to list plugins next to their doc strings:

```
$ idem doc group --output=json | jq -r '. | keys[] as $k | "\($k): \(.[$k].doc)"'
```

At the time of writing, with no extra plugins installed, this is the result of that command:

```
group.duration.apply: Sort the output by the total seconds of the state's run time
group.init.apply: Apply all group plugins specified in config
group.number.apply: Sort the data by run number
group.omit_noop.apply: Remove states that reported success without changes
group.sort.apply: Reorganize the data by sorting by each state's unique tag
```

26.2 Using Group Plugins

Group plugins are specified in a single string and are separated by a pipe. Each group plugin will be run in the order that it is defined.

Group plugins can be specified on the cli with the "–group" flag.

```
$ idem state my_state.sls --group="number|omit_noop"
```

Group plugins can also be specified in the idem config.

```
# idem.cfg
idem:
  group: number|omit_noop
```

26.3 Creating a Group Plugin

First create a directory at my_project_root/my_provider/group. In your project's conf.py, extend idem's namespace with your "group" directory.

```
# my_project_root/my_provider/conf.py
DYNE = {"group": ["group"]}
```

Now create a plugin in your "source" directory.

The plugin simply needs to implement the *apply* function. This function receives the *hub*, and *data* arguments. These will be passed into the function.

data contains the full results of a state run. Here are the contents of *data* for a run containing a single state:

```
{
    "resource_ref_|-state_block_name_|-state_name_|-function": {
        "tag": "resource_ref_|-state_block_name_|-state_name_|-function",
        "name": "state_name",
        "changes": {},
        "new_state": {},
        "old_state": {},
        "comment": None,
        "rerun_data": None,
        "result": True,
        "esm_tag": "resource_ref_|-state_block_name_|-state_name_|-",
        "__run_num": 1,
        "start_time": "2022-08-15 09:39:33.608291",
        "total_seconds": 0.001207,
        "sls_meta": {"SLS": {}, "ID_DECS": {}},
    },
}
```

This is what a basic group plugin looks like:

```
# my_project_root/my_provider/group/my_plugin.py
from typing import Any
from typing import Dict


def apply(hub, data: Dict[str, Any]) -> Dict[str, Any]:
    """
    Re-organize/filter the state runtime results from "data"
    """
    # initialize an ordered dictionary for the return (All dictionaries are ordered␣
↪after python 3.7)
    ret = {}
    # iterate over the state return data in the order you want to add it to the return
    for tag in data:
        # retrieve the result of a single state
        state_ret = data[tag]
        # Break the state tag into it's component parts
        comps = tag.split("_|-")
        state = comps[0]
```

(continues on next page)

(continued from previous page)

```
        id_ = comps[1]
        fun = comps[3]
        # "result" is True if the state ran successfully, otherwise it is False
        result = state_ret.get("result")
        # Any comment(s) from the running state
        comment = state_ret.get("comment")
        # An empty dictionary if there were no changes, else a comparison of "new" and
↪"old" state of the resource
        changes = state_ret.get("changes", {})
        # The state of the resource before the function ran
        old_state = state_ret.get("old_state", {})
        # The state of the resource after the function ran
        new_state = state_ret.get("new_state", {})

        # omit this state ret from the return data based on any of the previous␣
↪information
        if not True:
            continue

        # Copy the state ret from the input data to the return data
        ret[tag] = data[tag]

    return ret
```

CHAPTER

TWENTYSEVEN

RECONCILIATION LOOP

Reconciliation loop re-executes the states until all states are successfully realized or no progress is made.

27.1 Reconciler Plugin

The reconciler plugin provided by idem is called 'basic'. Idem reconciliation loop will re-apply pending states. It stops if none of the states is "pending" or if results/changes have not changed during the last 3 iterations.

27.2 Loop Implementation

To implement a reconciler plugin, provide a method like this:

```
from typing import Dict, Any

async def loop(
    hub,
    pending_plugin; str,
    name: str,
    apply_kwargs: Dict[str, Any],
    ):
    ...
```

The reconciler loop should return a dictionary like this:

```
{
    "re_runs_count": <number reconciliation loop iterations>,
    "require_re_run": <True or False>,
}
```

27.3 Reconciliation Wait Time

Reconciliation wait time is the sleep time between loop iteration. Each state can define a separate reconciliation wait time in seconds. For each iteration of the reconciliation loop, the wait time is re-calculated for the pending states, and the longest wait time value is used. By default, the idem reconciler plugin uses 3 seconds wait time unless defined by the state.

Idem supports three algorithms for calculating the wait time:

- static
- random
- exponential

27.3.1 Static

Fixed wait time value defined on the state, which remains the same for all iterations. For example:

```
__reconcile_wait__ = {"static": {"wait_in_seconds": 10}}
```

The default value is static of 3 seconds.

27.3.2 Random

Random wait time expects minimum and maximum values and generates a random number in that range (inclusive). It is defined in the state like this:

```
__reconcile_wait__ = {"random": {"min_value": 1, "max_value": 10}}
```

In this example, a random number from 1 to 10 is generated before each reconciliation iteration.

27.3.3 Exponential

A wait time that increases for every reconciliation iteration. The exponential wait time is calculated based on this formula:

wait_in_seconds * (multiplier ^ run_count)

Where '`run_count`' is the number of the iteration. The default value is 0. For example:

```
__reconcile_wait__ = {"exponential": {"wait_in_seconds": 2, "multiplier": 10}}
```

In the example, exponential wait times are: 2, 20, 200…

27.4 Pending plugin

Pending plugin is used to determine whether a state is in a "pending" state that requires reconciliation, which would re-apply the state.

To implement a pending plugin, provide a method like this:

```
def is_pending(hub, ret):
```

is_pending returns True if more reconciliation is needed, otherwise False.

The default implementation is defined in default.py, and returns False if '`result=True`' and there are no '`changes`', where '`changes`' is the delta between the previous state and the required state.

27.5 CLI

The reconciler plugin and pending plugin are specified as an argument to the `idem state` CLI.

For example:

```
--reconciler=basic | -R=basic | -r=basic
--pending=default | -P=default | ip=default
```

Reconciliation loop is enabled by default, to disable it use –reconciler=none.

27.6 Batch Function

Reconciliation loop can also be run in a batch command. For example:

```
hub.pop.Loop.run_until_complete(
    hub.idem.state.batch(
        states=states,
        name=name,
        runtime="serial",
        renderer="json",
        test=False,
        encrypted_profiles=encrypted_profiles,
        acct_key=acct_key,
        default_acct_profile="default",
        reconcile_plugin="basic",
        pending_plugin="default",
    )
)
```

27.7 Notes

- There is no reconciliation for exec commands (exec.run) that are successful.
- There is no reconciliation during 'test' invocations (–test=True).

CHAPTER

TWENTYEIGHT

ENFORCED STATE MANAGEMENT

Enforced state management (ESM) lets Idem track resources across runs. ESM makes it possible for resources that aren't natively idempotent to become idempotent through their unique present state name.

In the given context, the previous state (old_state) will be enforced with the following logic:

- Parameters for a resource in the given SLS file will have the highest priority
- If a parameter is not defined in the SLS, it will be pulled from the *old_state* of the previous run
- If there is no *old_state* or the parameter is not in *old_state*, then the default from the python function header will be used.

28.1 Local cache

The default ESM plugin keeps a local cache of the enforced state. The local cache is based on the `--root-dir`, `--cache-dir`, and `--run-name` cli arguments. Alternatively the `root_dir`, `cache_dir`, and `run_name` variables can be set in idem's config. The default root_dir is `/` when running idem as root, otherwise it is `~/.idem` . The default cache_dir is `cache` under `root_dir/var`; see below for examples:

If the run_name is `cli` (the default), then the `local` ESM plugin will store it's cache in `~/.idem/var/cache/idem/esm/local/cli.mspgack`. The ESM cache would be in `~/.idem/var/cache/idem/esm/cache/cli.mspgack`.

The cache contains the "new_state" data of state runs. Every key in the cache is a tag based on the state name, the state id, and the the state's name.

For a state that looks like this:

```
state_id:
    cloud.resource.present:
       name: state_name
```

The tag generated to track that state's "new_state" in the cache would look like this:

```
``cloud.resource_|-state_id_|-state_name_|-``
```

28.2 Idem states

State modules that return "old_state" and "new_state" will have "new_state" available in the `ctx` of future runs.

```
# my_project_root/my_project/state/my_plugin.py

__contracts__ = ["resource"]


def present(hub, ctx, name):
    # ctx.old_state contains the new_state from the previous run
    # When ctx.test is True, there should be no changes to the resource, but old_state␣
↪and new_state should reflect changes that `would` be made.
    new_state = ...

    return {
        "result": True,
        "comment": "",
        "old_state": ctx.old_state,
        "new_state": new_state,
    }


def absent(hub, ctx, name):
    # ctx.old_state contains the new_state from the previous run
    return {"result": True, "comment": "", "old_state": ctx.old_state, "new_state": {}}


def describe(hub, ctx, name):
    ...
```

28.3 Unlock Idem state run

When a state is run using an esm provider other than the local default (such as AWS) a lock may be left behind when the state is prematurely canceled. To force an unlock in this situation use a command line such as the following:

```
idem exec esm.unlock provider=aws profile=<...> --acct-file=<...> --acct-key=<...>
```

28.4 context

The context feature allows only one instance of an Idem state run for a given context. It also exposes a state dictionary that can be managed by an arbitrary plugin. The context is managed for you by Idem when you write an ESM plugin. The following shows how context works and how to use it:

```
async def my_func(hub):
    # Retrieve the context manager
    context_manager = hub.idem.managed.context(
        run_name=hub.OPT.idem.run_name,
        cache_dir=hub.OPT.idem.cache_dir,
```

(continues on next page)

(continued from previous page)

```
        esm_plugin="my_esm_plugin",
        esm_profile=hub.OPT.idem.esm_profile,
        acct_file=hub.OPT.acct.acct_file,
        acct_key=hub.OPT.acct.acct_key,
        serial_plugin=hub.OPT.idem.serial_plugin,
    )

    # Enter the context and lock the run.
    # This calls `hub.esm.my_esm_plugin.enter()` and `hub.esm.my_esm_plugin.get_state()`␣
↪with the appropriate ctx
    async with context_manager as state:
        # The output of get_state() is now contained in the "state" variable
        # Changes to `state` will persist when we exit the context and `hub.esm.my_esm_
↪plugin.set_state()` is called with the appropriate ctx
        state.update({})
    # After exiting the context, `hub.esm.my_esm_plugin.exit_() is called with the␣
↪appropriate ctx
```

28.5 Writing an ESM plugin

An ESM plugin follows this basic format:

```
# my_project_root/my_project/esm/my_plugin.py
from typing import Any
from typing import Dict


def __init__(hub):
    hub.esm.my_plugin.ACCT = ["my_acct_provider"]


async def enter(hub, ctx):
    """
    :param hub:
    :param ctx: A namespace addressable dictionary that contains the `acct` credentials
        "acct" contains the esm_profile from "my_acct_provider"

    Enter the context of the enforced state manager
    Only one instance of a state run will be running for the given context.
    This function enters the context and locks the run.

    The return of this function will be passed by Idem to the "handle" parameter of the␣
↪exit function
    """


async def exit_(hub, ctx, handle, exception: Exception):
    """
    :param hub:
    :param ctx: A namespace addressable dictionary that contains the `acct` credentials
```

(continues on next page)

(continued from previous page)

```
        "acct" contains the esm_profile from "my_acct_provider"
    :param handle: The output of the corresponding "enter" function
    :param exception: Any exception that was raised while inside the context manager or␣
↪None

    Exit the context of the Enforced State Manager
    """


async def get_state(hub, ctx) -> Dict[str, Any]:
    """
    :param hub:
    :param ctx: A dictionary with 3 keys:
        "acct" contains the esm_profile from "my_acct_provider"

    Use the information provided in ctx.acct to retrieve the enforced managed state.
    Return it as a python dictionary.
    """


async def set_state(hub, ctx, state: Dict[str, Any]):
    """
    :param hub:
    :param ctx: A namespace addressable dictionary that contains the `acct` credentials
        "acct" contains the esm_profile from "my_acct_provider"

    Use the information provided in ctx.acct to upload/override the enforced managed␣
↪state with "state".
    """
```

Extend the ESM dyne in your project for a plugin:

```
# my_project_root/my_project/conf.py
DYNE = {"esm": ["esm"]}
```

28.6 refresh

Idem includes a `refresh` command that can bring resources from `describe` into the ESM context.

The `refresh` command:

```
$ idem refresh aws.ec2.*
```

Is functionally equivalent to these commands:

```
$ idem describe aws.ec2.* --output=yaml > ec2.sls
$ idem state ec2.sls --test
$ rm ec2.sls
```

An `idem refresh` only returns resource attribtues and should not make any changes to resources as long as the resources implement the ctx.test flag properly in their present state.

28.7 restore

ESM keeps a cache of the local run state. The `restore` command calls an ESM plugin's `set_state` method with the contents of a json file.

```
$ idem restore esm_cache_file.json
```

The cache file is generated on every Idem state run and is based on Idem's `run_name` and `cache_dir`:

```
$ idem state my_state --run-name=my_run_name --cache-dir=/var/cache/idem
# Cache file for this run will be located in /var/cache/idem/my_run_name.json
```

CHAPTER

TWENTYNINE

PROGRESS BAR

Idem can show a progress bar for states. The progress bar displays one tick mark for each state completed, regardless of success or failure. Progress bars are printed to stderr and do not interfere with parsing program output. The default behavior is to show the progress bar.

29.1 Configuration

The config file supports the following options for progress bars:

```
# my_config.cfg

idem:
  # Always show a progress bar
  progress: True
  # The progress plugin to use, currently only "tqdm" is available
  progress_plugin: tqdm
  # kwargs to pass to the progress bar plugin "create" function
  progress_options:
    colour: green
```

29.2 CLI

The progress bar is enabled on the cli by default for the following cli commands:

```
$ idem state my_state.sls
```

```
$ idem describe my_resource
```

You can also explicitly disable the progress bar on the cli using the `--no-progress` flag.

```
$ idem state my_state.sls --no-progress
```

29.3 Examples

To add a progress bar, create and call an SLS file such as progress.sls. Use the following examples as guidelines.

29.3.1 Basic progress bar

The following progress.sls file creates a simple progress bar:

```
# progress.sls

{% for i in range(100) %}
sleep_{{ i }}:
  time.sleep:
    - duration: .1
{% endfor %}
```

To show the progress bar, add the following command line options. Use serial runtime to clearly display incremental tick marks. Parallel runtime is instantaneous and won't show a slow progression of tick marks:

```
$ idem state progress.sls --progress --runtime=serial
```

Progress bar output:

```
idem runtime: 0: 100%|¦¦¦¦¦¦¦¦¦¦¦¦| 100/100 [00:10<00:00,  9.63states/s]
```

29.3.2 Reconciliation

Idem reconciliation loops result in a progress bar for each loop as shown in the following example.

For demonstration purposes in the example, the first run is coded to fail in the nop state requisites. The failure then causes three reconciliation passes, each with its own progress bar:

```
# progress.sls

fail:
  test.present:
    - result: False

nop:
  test.nop:
    - require:
        - fail
```

```
$ idem state progress.sls --progress --runtime=serial
```

Progress bar output:

```
idem runtime: 0: 100%|¦¦¦¦¦¦¦¦¦¦¦¦| 2/2 [00:00<00:00, 959.79states/s]
idem runtime: 1: 100%|¦¦¦¦¦¦¦¦¦¦¦¦| 1/1 [00:00<00:00, 266.02states/s]
idem runtime: 2: 100%|¦¦¦¦¦¦¦¦¦¦¦¦| 1/1 [00:00<00:00, 264.14states/s]
idem runtime: 3: 100%|¦¦¦¦¦¦¦¦¦¦¦¦| 1/1 [00:00<00:00, 274.86states/s]
```

29.3.3 Displaying separate progress bars

The `unique` requisite creates a new runtime for each use of the `unique` requisite, which results in a new progress bar for each state.

```
# progress.sls

{% for i in range(10) %}
sleep_{{ i }}:
  time.sleep:
    - duration: .1
    - unique:
        - time.sleep
{% endfor %}
```

```
$ idem state progress.sls --progress --runtime=serial
```

Progress bar output:

```
idem runtime: 0:  10%|¦         | 1/10 [00:00<00:00,  9.80states/s]
idem runtime: 0:  10%|¦         | 1/10 [00:00<00:00,  9.46states/s]

idem runtime: 1:  11%|¦         | 1/9 [00:00<00:00,  9.56states/s]
idem runtime: 1:  11%|¦         | 1/9 [00:00<00:00,  9.23states/s]
idem runtime: 2:  12%|¦?        | 1/8 [00:00<00:00,  9.65states/s]
idem runtime: 2:  12%|¦?        | 1/8 [00:00<00:00,  9.33states/s]

idem runtime: 3:  14%|¦?        | 1/7 [00:00<00:00,  9.66states/s]
idem runtime: 3:  14%|¦?        | 1/7 [00:00<00:00,  9.32states/s]
idem runtime: 4:  17%|¦?        | 1/6 [00:00<00:00,  9.65states/s]
idem runtime: 4:  17%|¦?        | 1/6 [00:00<00:00,  9.43states/s]

idem runtime: 5:  20%|¦¦        | 1/5 [00:00<00:00,  9.70states/s]
idem runtime: 5:  20%|¦¦        | 1/5 [00:00<00:00,  9.39states/s]
idem runtime: 6:  25%|¦¦¦       | 1/4 [00:00<00:00,  9.63states/s]
idem runtime: 6:  25%|¦¦¦       | 1/4 [00:00<00:00,  9.29states/s]

idem runtime: 7:  33%|¦¦¦?      | 1/3 [00:00<00:00,  9.66states/s]
idem runtime: 7:  33%|¦¦¦?      | 1/3 [00:00<00:00,  9.35states/s]
idem runtime: 8:  50%|¦¦¦¦¦     | 1/2 [00:00<00:00,  9.65states/s]
idem runtime: 8:  50%|¦¦¦¦¦     | 1/2 [00:00<00:00,  9.35states/s]

idem runtime: 9: 100%|¦¦¦¦¦¦¦¦¦¦| 1/1 [00:00<00:00,  9.63states/s]
idem runtime: 9: 100%|¦¦¦¦¦¦¦¦¦¦| 1/1 [00:00<00:00,  9.40states/s]
```

29.4 Progress bars in PyCharm

Some terminal consoles, such as the PyCharm "Run" window, do not flush or erase a line of text.

To correctly display progress bars in PyCharm, edit your Run configuration, and enable the "emulate terminal in output console" option.

CHAPTER

THIRTY

COUNT

Count allows you to create n number of instances of the same resource.

To create identical states, use names for state replication. The following example shows how to create four internet gateway resources that start with `igw-` in the name.

SLS

```
igw-12345:
  aws.ec2.internet_gateway.present:
  - names:{% for i in range(4) %}
            Test {{ loop.index }}: {{loop.index}}
    {% endfor %}
  - tags:
    - Key: new-name
      Value: igw-9cd387e7
```

Result

```
      ID: igw-12345
Function: aws.ec2.internet_gateway.present
  Result: True
 Comment: Created 'igw-b440d01c'
 Changes: new:
    ----------
    Attachments:
    InternetGatewayId:
        igw-b440d01c
    OwnerId:
        000000000000
    Tags:
        |_
          ----------
          Key:
              new-name
          Value:
              igw-9cd387e7
```

```
      ID: igw-12345
Function: aws.ec2.internet_gateway.present
  Result: True
 Comment: Created 'igw-bc2861d1'
```

(continues on next page)

(continued from previous page)

```
 Changes: new:
    ----------
    Attachments:
    InternetGatewayId:
        igw-bc2861d1
    OwnerId:
        000000000000
    Tags:
        |_
          ----------
          Key:
              new-name
          Value:
              igw-9cd387e7
```

```
      ID: igw-12345
Function: aws.ec2.internet_gateway.present
  Result: True
 Comment: Created 'igw-7bf9255d'
 Changes: new:
    ----------
    Attachments:
    InternetGatewayId:
        igw-7bf9255d
    OwnerId:
        000000000000
    Tags:
        |_
          ----------
          Key:
              new-name
          Value:
              igw-9cd387e7
```

```
      ID: igw-12345
Function: aws.ec2.internet_gateway.present
  Result: True
 Comment: Created 'igw-3aba95d8'
 Changes: new:
    ----------
    Attachments:
    InternetGatewayId:
        igw-3aba95d8
    OwnerId:
        000000000000
    Tags:
        |_
          ----------
          Key:
              new-name
          Value:
              igw-9cd387e7
```

To create states with different properties, use a jinja template in a loop. The following example shows how to create three subnets, where each subnet belongs to a different availability zone:

SLS

```
{% set aws_availability_zones = {"available": { "names": ["us-east-2a", "us-west-2b",
↪"eu-west-3"]}}%}
{% set VpcSuperNet = "10.0."%}
{% for i in range(3) %}
aws_subnet.cluster-{{i}}:
  aws.ec2.subnet.present:
  - availability_zone: {{aws_availability_zones.available.names[i]}}
  - vpc_id: vpc-3d44da2d
  - cidr_block: {{VpcSuperNet+(i | string)}}.0/18
  - tags: {{ [{"Key":"Name", "Value":"test-"+(i | string)} ] }}
{% endfor %}
```

Result

```
      ID: aws_subnet.cluster-0
Function: aws.ec2.subnet.present
  Result: True
 Comment: Created 'aws_subnet.cluster-0'
 Changes: new:
    ----------
    name:
        aws_subnet.cluster-0
    resource_id:
        subnet-d7cd43a1
    vpc_id:
        vpc-3d44da2d
    cidr_block:
        10.0.0.0/18
    availability_zone:
        us-west-2a
    tags:
        |_
          ----------
          Key:
              Name
          Value:
              test-0
```

```
      ID: aws_subnet.cluster-1
Function: aws.ec2.subnet.present
  Result: True
 Comment: Created 'aws_subnet.cluster-1'
 Changes: new:
    ----------
    name:
        aws_subnet.cluster-1
    resource_id:
        subnet-ad763648
    vpc_id:
```

(continues on next page)

(continued from previous page)

```
        vpc-3d44da2d
    cidr_block:
        10.0.1.0/18
    availability_zone:
        us-west-2b
    tags:
        |_
          ----------
          Key:
              Name
          Value:
              test-0
```

```
      ID: aws_subnet.cluster-2
Function: aws.ec2.subnet.present
  Result: True
 Comment: Created 'aws_subnet.cluster-2'
 Changes: new:
    ----------
    name:
        aws_subnet.cluster-2
    resource_id:
        subnet-bc438686
    vpc_id:
        vpc-3d44da2d
    cidr_block:
        10.0.2.0/18
    availability_zone:
        us-west-2c
    tags:
        |_
          ----------
          Key:
              Name
          Value:
              test-0
```

CHAPTER

THIRTYONE

EVENTS

Every event follows a predictable format:

```
{
    "tags": {
        "ref": "A reference to the function on the hub that fired this event",
        "type": "An identifier to describe the nature of the message"
    },
    "message": "Message data, which can be any serializable object",
    "run_name": "The user-given run_name"
}
```

31.1 Firing Events

31.1.1 from code

The `body` is any serializable data that comprises the main part of the event The `profile` is the ingress profile from `acct` that this event should be published to.

Asynchronous put:

```
async def my_func(hub):
    await hub.idem.event.put(
        body="Any serializable object",
        profile="idem-[plugin]",
        tags={},
    )
```

Synchronous put:

```
def my_func(hub):
    hub.idem.event.put_nowait(
        body="Any serializable object",
        profile="idem-[plugin]",
        tags={},
    )
```

31.1.2 from jinja/sls

Events can also be fired from within an idem sls file via jinja:

```
{%- hub.idem.event.put_nowait(body={"message": "event content"}, profile="default", tags=
↪{}) %}
```

31.2 Event Profiles

Events in idem are published to profiles of a specific name. Create an event profile associated with specific events to subscribe to that event with your chosen provider. Multiple providers can be configured for the same event profile.

```
kafka:
  event_profile_name:
    connection:
      bootstrap_servers: localhost:9092
pika:
  event_profile_name:
    connection:
      host: localhost
      port: 5672
      login: guest
      password: guest
```

A profile name can be specified multiple times within the same provider.

```
kafka:
  - event_profile_name:
      connection:
        bootstrap_servers: localhost:9092
  - event_profile_name:
      connection:
        bootstrap_servers: my_server:9092
```

31.2.1 idem-*

Create a profile called `idem-*` to receive ALL events from idem.

```
my_provider:
  idem-*:
    provider_connection_data:
```

The default plugin for this matching is `glob`. A different acct_file wide `match_plugin` can be specified by adding a `match_plugin` keyword to your acct_file. Read more about match plugins in pop-evbus.

```
match_plugin: glob|regex
my_provider:
  idem-*:
    provider_connection_data:
```

31.2.2 idem-status

Create an evbus provider profile called `idem-status` to receive events about the status of the current run.

```
my_provider:
  idem-status:
    provider_connection_data:
```

Message body format for status data:

```
{
    "tags": {"ref": "idem.state.update_status", "type": "state-status"},
    "message": "FINISHED/CREATED/GATHERING/COMPILING/RUNNING/COMPILATION_ERROR/GATHER_
↪ERROR/RUNTIME_ERROR/UNDEFINED",
    "run_name": "The user supplied run-name"
}
```

31.2.3 idem-low

Create an evbus provider profile called `idem-low` to receive events about sls low data.

```
my_provider:
  idem-low:
    provider_connection_data:
```

Message body format for low data:

```
{
    "tags": {"ref": "idem.run.init.start", "type": "state-low-data"},
    "message": [
        {
            "name": "Name of the state",
            "state": "Reference on the hub to state plugin",
            "fun": "The state function name",
            "__sls__": "The sls source",
            "__id__": "The state id, usually it will be the same as name",
            "order": 100000
        }
    ],
    "run_name": "The user supplied run-name"
}
```

31.2.4 idem-high

Create an evbus provider profile called `idem-high` to receive events about sls rendered high data.

```
my_provider:
  idem-high:
    provider_connection_data:
```

Message body format for high data:

```
{
    "message": {
        "Reference on the hub to the state plugin": {
            "__sls__": "The stem/name of the sls source",
            "The reference to the state plugin": ["The reference to the state function"]
        }
    },
    "run_name": "The user supplied run-name",
    "tags": {"ref": "idem.sls_source.init.gather", "type": "state-high-data"}
}
```

31.2.5 idem-state

Create an evbus provider profile called `idem-state` to receive the pre/post state information.

```
my_provider:
  idem-state:
    provider_connection_data:
```

pre

Message body format for run data:

```
{
    "message": {
        "Name of the state": {
            "The reference to the state function": {
                "ctx": {"run_name": "The run_name specified on the cli", "test": false},
                "kwargs": {},
                "name": "Name of the state"
            }
        }
    },
    "run_name": "The user supplied run-name",
    "tags": {
        "ref": "Reference on the hub to the state function that fired the event",
        "type": "state-pre",
        "acct_details": "Information that can link this event to acct credentials in the␣
↪calling function"
    }
}
```

post

Message body format for run data:

```
{
        "message": {
            "changes": {"old": [], "new": []},
            "comment": "",
            "name": "Name of the state",
            "result": true
        },
        "run_name": "The user supplied run-name",
        "tags": {"ref": "Reference on the hub to the state function that fired the event
↪", "type": "state-post",
        "acct_details": "Information that can link this event to acct credentials in the␣
↪calling function"
    }
}
```

31.2.6 idem-chunk

Create an evbus provider profile called `idem-chunk` to receive individual fully compiled states.

```
my_provider:
  idem-chunk:
    provider_connection_data:
```

Message body format for run data:

```
{
        "message": {
            "name": "Name of the state",
            "state": "Reference on the hub to the state plugin",
            "fun": "Reference on the hub to the state function",
            "__id__": "The state id, usually same as name",
            "__sls__": "Tye sls source",
            "order": 100000
        },
        "run_name": "The user supplied run-name",
        "tags": {"ref": "Reference on the hub to the state function that fired the event
↪", "type": "state-post"
        }
}
```

31.2.7 idem-run

Create an evbus provider profile called `idem-run` to receive the output of each state with complete meta-data

```
my_provider:
  idem-run:
    provider_connection_data:
```

Message body format for run data:

```
{
        "message": {
            "name": "Name of the state",
            "__id__": "The state id",
            "order": "An integer that helps idem determine which states to run first",
            "__run_num": "The run number of the state",
            "changes": "A dictionary of changes made in the state",
            "comment": "A comment supplied by the state",
            "esm_tag": "The key used to store this state in the ESM cache",
            "tag": "The key used to store this state in the RUNS internal structure",
            "new_state": "The state of a resource after a run",
            "old_state": "The state of a resource before a run",
            "result": "True if the state ran successfully, else False"
        },
        "run_name": "The user supplied run-name",
        "tags": {"ref": "idem.rules.init.run", "type": "state-result",
        "acct_details": "Information that can link this event to acct credentials in the␣
↪calling function"
}
}
```

31.2.8 idem-exec

Create an evbus provider profile called `idem-exec` to receive the returns of all idem exec modules as events.

```
my_provider:
  idem-exec:
    provider_connection_data:
```

Message body format for exec data:

```
{
        "message": {"result": true, "ret": "Any object", "comment": "Any string"},
        "run_name": "The user supplied run-name",
        "tags": {
            "type": "exec-post",
            "ref": "A reference to the function on the hub that fired this event",
            "acct_details": "Information that can link this event to acct credentials in␣
↪the calling function"
        }
}
```

31.2.9 logger

Create an evbus profile called `idem-logger` to receive all log messages from pop as events.

```
my_provider:
  idem-logger:
    provider_connection_data:
```

When starting idem from the command line, be sure to specify `--log-handler=event`.

```
idem state state.sls --log-level=debug --log-handler=event
```

Message body format for logs:

```
{
        "message": "The log message",
        "run_name": "The user supplied run-name",
        "tags": {
            "module": "module that produced the log",
            "level": "log level name",
            "timestamp": "asctime timestamp",
            "ref": "A reference to the function on the hub that fired this event"
        }
}
```

CHAPTER

THIRTYTWO

KUBERNETES CRD SUPPORT

Idem supports using Kubernetes CRD to execute state. The kubernetes CRD is internally converted into SLS format used by idem.

32.1 CRD format

The CRD format is similar to kubernetes syntax. Following is a sample CRD SLS file:

```
apiVersion: resource-management.azure.idem.vmware.com/v1alpha1
kind: resource-groups
metadata:
  name: new-rg
spec:
  - resource_group_name: new-rg
  - parameters:
      location: eastus
      tags:
        env: new-rg
        Unit: CMBU
```

The above CRD gets converted internally into the following SLS:

```
new-rg:
  azure.resource_management.resource_groups.present:
  - resource_group_name: new-rg
  - parameters:
    location: eastus
    tags:
      env: new-rg
      Unit: CMBU
```

As clear from the above example:

1. `apiVersion` together with `kind` attribute identifies the *path reference* of the SLS.
2. `metadata`'s `name` is the state id of the SLS.
3. `spec` contains any parameters required by the resulting state.
4. *function reference* is always *present* by default, but can be inverted using command-line parameter `--invert`.

32.2 Execution

To execute a Kubernetes CRD, command-line parameter `--render 'jinja|yaml|k8crd'` needs to be specified.

Following example details a *resource group* creation using CRD as mentioned in previous section:

```
$ idem state --output json --render 'jinja|yaml|k8crd' crd.sls
{
    "azure.resource_management.resource_groups_|-new-rg_|-new-rg_|-present": {
        "changes": {
            "new": {
                "id": "/subscriptions/subscription-id/resourceGroups/new-rg",
                "name": "new-rg",
                "type": "Microsoft.Resources/resourceGroups",
                "location": "eastus",
                "tags": {
                    "env": "new-rg",
                    "Unit": "CMBU"
                },
                "properties": {
                    "provisioningState": "Succeeded"
                }
            }
        },
        "comment": "Created",
        "name": "new-rg",
        "result": true,
        "old_state": null,
        "new_state": null,
        "__run_num": 1
    }
}
```

Following example details *resouce group* deletion using the same CRD:

```
$ idem state --output json --render 'jinja|yaml|k8crd' --invert crd.sls
{
    "azure.resource_management.resource_groups_|-new-rg_|-new-rg_|-absent": {
        "changes": {
            "old": {
                "id": "/subscriptions/subscription-id/resourceGroups/new-rg",
                "name": "new-rg",
                "type": "Microsoft.Resources/resourceGroups",
                "location": "eastus",
                "tags": {
                    "env": "new-rg",
                    "Unit": "CMBU"
                },
                "properties": {
                    "provisioningState": "Succeeded"
                }
            }
        },
```

(continues on next page)

(continued from previous page)

```
        "comment": "Accepted",
        "name": "new-rg",
        "result": true,
        "old_state": null,
        "new_state": null,
        "__run_num": 1
    }
}
```

CHAPTER

THIRTYTHREE

IDEM SCRIPTS

Idem runs can be initialized from a python script. If at all possible, it is ideal to use idem's CLI. However, in some FAAS applications shelling out isn't always an option; for example aws lambdas must use pure python. The following examples show how to run idem from a pure python script.

In this example, states are run with the minimum configuration:

```
import pop.hub
import json

# Create the hub
hub = pop.hub.Hub()
# Add idem's dynamic namespace to the hub, which loads all idem-related subs onto the hub
# I.E. states/exec/tool/acct/etc...
# All python projects in the current python environment that have a conf.py with a DYNE␣
↪dictionary will be loaded
# They will be loaded onto subs based on the mappings in the DYNE dictionary
hub.pop.sub.add(dyne_name="idem")

# Set up variables to use in the run
run_name = "my_run"
sls_name = "my_sls"
test = False
invert_state = False
acct_profile = "default"
cache_dir = "/dev/null"
runtime = "parallel"
params = []
param_sources = []
esm_plugin = "local"
esm_profile = "default"

# Compile states as a dictionary
my_states = {
    f"{sls_name}.sls": {
        "state_name": {
            "test.nop": [
                {"name": "value"},
                {"kwarg1": "value1"},
            ]
        }
    }
```

(continues on next page)

(continued from previous page)

```
}

# Compile acct data that will be used for the run
acct_data = {
    "profiles": {
        "provider_name": {
            "profile_name": {
                "kwarg_1": "value_1",
                "kwarg_2": "value_2",
            },
            "default": {
                "kwarg_1": "value_1",
                "kwarg_2": "value_2",
            },
        }
    }
}

# Create the event loop
hub.pop.loop.create()
# Run the states
hub.pop.Loop.run_until_complete(
    hub.idem.state.apply(
        name=run_name,
        sls_sources=[f"json://{json.dumps(my_states)}"],
        render="json",
        runtime=runtime,
        subs=["states"],
        cache_dir=cache_dir,
        sls=[sls_name],
        test=test,
        invert_state=invert_state,
        acct_profile=acct_profile,
        acct_data=acct_data,
        managed_state={},
        param_sources=param_sources,
        params=params,
    )
)

# Gather the results from RUNS
results = hub.idem.RUNS[run_name]["running"]
errors = hub.idem.RUNS[run_name]["errors"]

# Do things with the resulting data
print(f"State compile errors: {errors}")
print(f"State run results: {results}")
```

This example runs states in an esm context:

```
import pop.hub
import json
```

(continues on next page)

(continued from previous page)

```
# Create the hub
hub = pop.hub.Hub()
# Add idem's dynamic namespace to the hub, which loads all idem-related subs onto the hub
# I.E. states/exec/tool/acct/etc...
# All python projects in the current python environment that have a conf.py with a DYNE␣
↪dictionary will be loaded
# They will be loaded onto subs based on the mappings in the DYNE dictionary
hub.pop.sub.add(dyne_name="idem")

# Set up variables to use in the run
run_name = "my_run"
sls_name = "my_sls"
test = False
invert_state = False
acct_profile = "default"
cache_dir = "/dev/null"
runtime = "parallel"
params = []
param_sources = []
esm_plugin = "local"
esm_profile = "default"

# Compile states as a dictionary
my_states = {
    f"{sls_name}.sls": {
        "state_name": {
            "test.nop": [
                {"name": "value"},
                {"kwarg1": "value1"},
            ]
        }
    }
}

# Compile acct data that will be used for the run
acct_data = {
    "profiles": {
        "provider_name": {
            "profile_name": {
                "kwarg_1": "value_1",
                "kwarg_2": "value_2",
            },
            "default": {
                "kwarg_1": "value_1",
                "kwarg_2": "value_2",
            },
        }
    }
}
```

(continues on next page)

(continued from previous page)

```
async def start():
    # Configure the context for ESM
    context_manager = hub.idem.managed.context(
        run_name=run_name,
        cache_dir=cache_dir,
        esm_plugin=esm_plugin,
        esm_profile=esm_profile,
        acct_data=acct_data,
    )
    # Run the states in the ESM context
    async with context_manager as state:
        await hub.idem.state.apply(
            name=run_name,
            sls_sources=[f"json://{json.dumps(my_states)}"],
            render="json",
            runtime=runtime,
            subs=["states"],
            cache_dir=cache_dir,
            sls=[sls_name],
            test=test,
            invert_state=invert_state,
            acct_profile=acct_profile,
            acct_data=acct_data,
            managed_state=state,
            param_sources=param_sources,
            params=params,
        )


# Create the event loop
hub.pop.loop.create()
# Run idem in the event loop
hub.pop.Loop.run_until_complete(start)

# Gather the results from RUNS
results = hub.idem.RUNS[run_name]["running"]
errors = hub.idem.RUNS[run_name]["errors"]

# Do things with the resulting data
print(f"State compile errors: {errors}")
print(f"State run results: {results}")
```

This example runs an exec module with the minimum configuration required:

```
import pop.hub
import json

# Create the hub
hub = pop.hub.Hub()
# Add idem's dynamic namespace to the hub, which loads all idem-related subs onto the hub
# I.E. states/exec/tool/acct/etc...
# All python projects in the current python environment that have a conf.py with a DYNE␣
```

(continues on next page)

(continued from previous page)

```
↪dictionary will be loaded
# They will be loaded onto subs based on the mappings in the DYNE dictionary
hub.pop.sub.add(dyne_name="idem")

# Use the run_name as a routing key for exec module events
hub.idem.RUN_NAME = run_name = "my_run"

# Compile acct data that will be used for the run
acct_profile = "default"
acct_data = {
    "profiles": {
        "provider_name": {
            "profile_name": {
                "kwarg_1": "value_1",
                "kwarg_2": "value_2",
            },
            "default": {
                "kwarg_1": "value_1",
                "kwarg_2": "value_2",
            },
        }
    }
}

# positional arguments for the exec module go here.
args = []
# Keyword arguments for teh exec module go here
kwargs = {}

# NOTE: hub and ctx should not be passed to the exec module.
# The hub is implicitly passed and ctx is constructed from acct_data

hub.pop.loop.create()
# Run the exec module in the loop
result = hub.pop.Loop.run_until_complete(
    hub.idem.ex.run(
        "test.ping",
        args=args,
        kwargs=kwargs,
        acct_data=acct_data,
        acct_profile=acct_profile,
    )
)

# Do things with the result
print(result.result)
print(result.comment)
print(result.ret)
```

CHAPTER

THIRTYFOUR

IDEM DESCRIBE

The `idem describe` command lists the resources in a cloud account.

The `idem describe` command supports state file paths and regular expressions as input. In addition, you can refine output by applying a filter.

34.1 State file path as input

The following command returns all AWS S3 buckets in the provided cloud account.

```
idem describe aws.s3.bucket
```

34.2 Regular expression as input

Any valid regular expression can be used as an input. The following command returns all AWS resources in the provided cloud account.

```
idem describe "aws.*"
```

The following command returns all dynamodb and S3 resources in the provided cloud account.

```
idem describe "aws\.(dynamodb|s3)\..*"
```

34.3 Filtering

To further refine output, you can also add the `--filter` argument.

The following command only returns automatically named S3 buckets that start with the word *production*.

```
idem describe aws.s3.bucket --filter="[?resource[?bucket_prefix=='production']]"
```

CHAPTER

THIRTYFIVE

TUTORIALS

35.1 Write To File Function

Idem supports writing any data to a file. The 'parameters' might include argument bindings to record details from resources referenced by the same Idem invocation. The function can accept a Jinja template or a template file, which can generate a script from the 'parameters'. If you omit a template, and the 'parameters' are a dictionary, they will be written to the file in a json format.

For example:

```
my-output-file:
    data.write:
    - file_name: "/files/audit.log"
    - template: '{% raw  %}
                   {% for resource_name, resource_id in parameters.items() %}
                        Created {{ resource_name }}  {{ resource_id }}
                   {% endfor %}
                {%  endraw %}'
   - parameters:
        ${resource-1:name}: ${resource-1:id}
        ${resource-2:name}: ${resource-2:id}
```

The preceding write function produces a */files/audit.log* text file that contains the following:

```
Created subnet-1 a7dd1f64-8e02-11ec-b909-0242ac120002
Created subnet-2 b08bcf20-8e02-11ec-b909-0242ac120002
```

35.2 Template Render Function

Idem supports rendering data from jinja template file or an embedded jinja template. The 'variables' includes key-value pairs which will be used for interpolation within the template. The function can accept a Jinja template or a template file.

For example:

```
my-rendered-data:
  template.render:
    - template: '{% raw %}Hello {{ name }} !!{% endraw %}'
    - variables:
        name: "World"
```

The preceding render function produces the below content:

```
Hello World !!
```

35.3 Sleep Function

The idem time.sleep function pauses idem execution before or after resource state enforcement. Use `require` to pause before enforcement or `require_in` to pause after enforcement. Set the `duration` argument to the desired pause time in seconds.

The following example delays enforcement of the `some_machine` resource:

```
sleep_60s:
  time.sleep:
    - duration: 60

some_machine:
  cloud.instance.present:
    - name: my-instance
    - require:
      - time.sleep: sleep_60s
```

The following example adds a delay after the `some_machine` resource is enforced:

```
sleep_60s:
  time.sleep:
    - duration: 60

some_machine:
  cloud.instance.present:
    - name: my-instance
    - require_in:
      - time.sleep: sleep_60s
```

35.4 Trigger State in Idem

Trigger state lets Idem define a dictionary of key-value(K-V) pairs. Keys are user-defined and Values can be output of any other state or any other value a user wishes to define.

Behaviour of Trigger state : - check the value in the old_state dictionary and compare with the current_state. - if there is difference between old_state and new_state , result[changes] is populated with the diff.

Trigger state can be seen as a helper state. It can be used in conjunction with other state.

35.4.1 Example

```
``
always-changes-and-succeeds:
      test.succeed_with_changes:
        - name: foo

    always-changes-trigger:
        trigger.build:
            - triggers:
                - last_run_id: {{ range(1, 51) | random }}
                - comment: ${test:always-changes-and-succeeds:testing}

    watch_changes:
      test.nop:
        - onchanges:
          - trigger: always-changes-trigger

``
```

In the above example , *always-changes-trigger* is a trigger state with a dictionary of K-V pairs. *watch_changes* is watching for *changes* of *always-changes-trigger*. Iff there are *changes* in the trigger state , state *watch_changes* will execute.

CHAPTER

THIRTYSIX

SINGLE TARGET

target CLI option allow you to specify a single state. The resource will be executed with all its dependencies. The target value is the declaration ID of the state, which is guaranteed to be unique. If there are multiple states under a single declaration ID all of them will be invoked.

idem state <SLS> –target=<the_target>

Specifying an invalid target will result in an error.

For example, for the following SLS:

resource_a:

type1.present:

- require: - resource_b

resource_b:

type2.present:

- name: the_name_of_resource_b
- require: - resource_c

resource_c:
type3.present

grouped_resource_d:

typeA.present:

- require: - resource_c

typeB.present:

- name: typeB_resource

typeC.present:

- name: typeC_resource

idem state SLS –target=resource_a

will result in resource_a, resource_b and resource_c.

idem state SLS –target=grouped_resource_d

will result in the 3 nested resources (typeA, typeB and typeC) and resource_c.

CHAPTER

THIRTYSEVEN

TUTORIALS

37.1 Example Tutorial

A friendly message about why idem is useful for this walkthrough.

CHAPTER

THIRTYEIGHT

MICROSOFT AZURE CLOUD PROVIDER

Setting up cloud resources using Idem is easy to do! Check out the documentation at the link below:

Azure for Idem (idem-azurerm) Documentation

CHAPTER

THIRTYNINE

MIGRATING SUPPORT FROM SALT

Idem is not too far from Salt States. Idem extends Salt State functionality though, and uses slightly different underlying interfaces. for instance, Idem does not use *__salt__* or any of the dunder constructs, all of this information is now on the hub. But migration is intended to be easy!

39.1 Exec Modules and State Modules

Idem follows the same constructs as Salt in seperating execution functionality from idempotent enforcement into two seperate subsystems. The idea is that these are seperate concerns and that raw execution presents value in itself making the code more reusable.

39.1.1 salt/modules to exec

Modules inside of *salt/modules* should be implemented as *exec* modules in Idem. References on the hub should be changed from *__salt__['test.ping']* to references on the hub, like *hub.exec.test.ping*.

39.1.2 salt/states to states

Modules inside of *salt/states* should be implemented as *states* modules in Idem. References on the hub should be changed from *__states__['pkg.installed']* to *hub.states.pkg.installed*.

39.1.3 salt/utils to exec

Many Salt modules use functions inside of utils. This grew in Salt out of limitations from the salt loader and how shared code was originally developed.

For Idem anything that is in utils should be moved into *exec*. This makes those functions generally available for everything else on the hub which solves the problem that created the utils system in Salt to begin with.

39.2 Namespaces

Unlike Salt's loader, POP allows for nested plugin subsystems. Idem recursively loads all lower subsystems for *exec* and *states* subsystems.

This means that you can move *exec* and *states* plugins into subdirectories! So when porting a module called *salt/modules/boto_s3.py* it could be ported to *exec/boto/s3.py*, or it could be ported to *exec/aws/s3.py* or *exec/aws/storage/s3.py*. The location of the file reflects the location on the hub, so these locations get referenced on the hub as *hub.exec.boto.s3*, *hub.exec.aws.s3*, *hub.exec.aws.storage.s3* respectively.

39.3 Exec Function Calls

All function calls now need to accept the hub as the first argument. Functions should also be changed to be async functions where appropriate. So this *exec* function signature:

```
def upload_file(
    source,
    name,
    extra_args=None,
    region=None,
    key=None,
    keyid=None,
    profile=None,
    ):
```

Gets changed to look like this:

```
async def upload_file(
    hub,
    source,
    name,
    extra_args=None,
    region=None,
    key=None,
    keyid=None,
    profile=None,
    ):
```

39.4 States Function Calls

States function calls now accept a *ctx* argument. This allows us to send an execution context into the function. The *ctx* is a dict with the keys *test* and *run_name*. The *test* value is a boolean telling the state if it is running is test mode. The *run_name* is the name of the run as it is stored on the hub, using the *run_name* you can gain access to the internal tracking data for the execution of the Idem run located in *hub.idem.RUNS[ctx['run_name']]*.

So a state function signature that looks like this in Salt:

```
def object_present(
    name,
    source=None,
```

(continues on next page)

(continued from previous page)

```
    hash_type=None,
    extra_args=None,
    extra_args_from_pillar='boto_s3_object_extra_args',
    region=None,
    key=None,
    keyid=None,
    profile=None):
```

Will look like this in Idem:

```
async def object_present(
    hub,
    ctx,
    name,
    source=None,
    hash_type=None,
    extra_args=None,
    extra_args_from_pillar='boto_s3_object_extra_args',
    region=None,
    key=None,
    keyid=None,
    profile=None):
```

39.5 Full Function Example

This example takes everything into account given a state function before and after. Doc strings are omitted for brevity but should be preserved.

39.5.1 Salt Function

```
def object_present(
    name,
    source=None,
    hash_type=None,
    extra_args=None,
    extra_args_from_pillar='boto_s3_object_extra_args',
    region=None,
    key=None,
    keyid=None,
    profile=None,
):
    ret = {
        'name': name,
        'comment': '',
        'changes': {},
    }

    if extra_args is None:
        extra_args = {}
```

(continues on next page)

(continued from previous page)

```
    combined_extra_args = copy.deepcopy(
        __salt__['config.option'](extra_args_from_pillar, {})
    )
    __utils__['dictupdate.update'](combined_extra_args, extra_args)
    if combined_extra_args:
        supported_args = STORED_EXTRA_ARGS | UPLOAD_ONLY_EXTRA_ARGS
        combined_extra_args_keys = frozenset(six.iterkeys(combined_extra_args))
        extra_keys = combined_extra_args_keys - supported_args
        if extra_keys:
            msg = 'extra_args keys {0} are not supported'.format(extra_keys)
            return {'error': msg}

    # Get the hash of the local file
    if not hash_type:
        hash_type = __opts__['hash_type']
    try:
        digest = salt.utils.hashutils.get_hash(source, form=hash_type)
    except IOError as e:
        ret['result'] = False
        ret['comment'] = "Could not read local file {0}: {1}".format(
            source,
            e,
        )
        return ret
    except ValueError as e:
        # Invalid hash type exception from get_hash
        ret['result'] = False
        ret['comment'] = 'Could not hash local file {0}: {1}'.format(
            source,
            e,
        )
        return ret

    HASH_METADATA_KEY = 'salt_managed_content_hash'
    combined_extra_args.setdefault('Metadata', {})
    if HASH_METADATA_KEY in combined_extra_args['Metadata']:
        # Be lenient, silently allow hash metadata key if digest value matches
        if combined_extra_args['Metadata'][HASH_METADATA_KEY] != digest:
            ret['result'] = False
            ret['comment'] = (
                'Salt uses the {0} metadata key internally,'
                'do not pass it to the boto_s3.object_present state.'
            ).format(HASH_METADATA_KEY)
            return ret
    combined_extra_args['Metadata'][HASH_METADATA_KEY] = digest
    # Remove upload-only keys from full set of extra_args
    # to create desired dict for comparisons
    desired_metadata = dict(
        (k, v) for k, v in six.iteritems(combined_extra_args)
        if k not in UPLOAD_ONLY_EXTRA_ARGS
    )
```

(continues on next page)

(continued from previous page)

```
    # Some args (SSE-C, RequestPayer) must also be passed to get_metadata
    metadata_extra_args = dict(
        (k, v) for k, v in six.iteritems(combined_extra_args)
        if k in GET_METADATA_EXTRA_ARGS
    )
    r = __salt__['boto_s3.get_object_metadata'](
        name,
        extra_args=metadata_extra_args,
        region=region,
        key=key,
        keyid=keyid,
        profile=profile,
    )
    if 'error' in r:
        ret['result'] = False
        ret['comment'] = 'Failed to check if S3 object exists: {0}.'.format(
            r['error'],
        )
        return ret

    if r['result']:
        # Check if content and metadata match
        # A hash of the content is injected into the metadata,
        # so we can combine both checks into one
        # Only check metadata keys specified by the user,
        # ignore other fields that have been set
        s3_metadata = dict(
            (k, r['result'][k]) for k in STORED_EXTRA_ARGS
            if k in desired_metadata and k in r['result']
        )
        if s3_metadata == desired_metadata:
            ret['result'] = True
            ret['comment'] = 'S3 object {0} is present.'.format(name)
            return ret
        action = 'update'
    else:
        s3_metadata = None
        action = 'create'

    def _yaml_safe_dump(attrs):
        '''
        Safely dump YAML using a readable flow style
        '''
        dumper_name = 'IndentedSafeOrderedDumper'
        dumper = __utils__['yaml.get_dumper'](dumper_name)
        return __utils__['yaml.dump'](
            attrs,
            default_flow_style=False,
            Dumper=dumper)

    changes_diff = ''.join(difflib.unified_diff(
        _yaml_safe_dump(s3_metadata).splitlines(True),
```

(continues on next page)

(continued from previous page)

```
        _yaml_safe_dump(desired_metadata).splitlines(True),
    ))

    if __opts__['test']:
        ret['result'] = None
        ret['comment'] = 'S3 object {0} set to be {1}d.'.format(name, action)
        ret['comment'] += '\nChanges:\n{0}'.format(changes_diff)
        ret['changes'] = {'diff': changes_diff}
        return ret

    r = __salt__['boto_s3.upload_file'](
        source,
        name,
        extra_args=combined_extra_args,
        region=region,
        key=key,
        keyid=keyid,
        profile=profile,
    )

    if 'error' in r:
        ret['result'] = False
        ret['comment'] = 'Failed to {0} S3 object: {1}.'.format(
            action,
            r['error'],
        )
        return ret

    ret['result'] = True
    ret['comment'] = 'S3 object {0} {1}d.'.format(name, action)
    ret['comment'] += '\nChanges:\n{0}'.format(changes_diff)
    ret['changes'] = {'diff': changes_diff}
    return ret
```

39.5.2 Idem State Function

```
async def object_present(
    hub,
    ctx,
    name,
    source=None,
    hash_type=None,
    extra_args=None,
    region=None,
    key=None,
    keyid=None,
    profile=None):
    ret = {
        'name': name,
        'comment': '',
```

(continues on next page)

(continued from previous page)

```
        'changes': {},
    }

    if extra_args is None:
        extra_args = {}
    # Pull out args for pillar

    # Get the hash of the local file
    if not hash_type:
        hash_type = hub.OPT['idem']['hash_type']  # Pull opts from hub.OPT
    try:
        # Some functions from utils will need to be ported over. Some general
        # Use functions should be sent upstream to be included in Idem.
        digest = hub.exec.utils.hashutils.get_hash(source, form=hash_type)
    except IOError as e:
        ret['result'] = False
        # Idem requires Python 3.6 and higher, use f-strings
        ret['comment'] = f'Could not read local file {source}: {e}'
        return ret
    except ValueError as e:
        # Invalid hash type exception from get_hash
        ret['result'] = False
        ret['comment'] = f'Could not hash local file {source}: {e}'
        return ret

    HASH_METADATA_KEY = 'idem_managed_content_hash'  # Change salt refs to idem
    combined_extra_args.setdefault('Metadata', {})
    if HASH_METADATA_KEY in combined_extra_args['Metadata']:
        # Be lenient, silently allow hash metadata key if digest value matches
        if combined_extra_args['Metadata'][HASH_METADATA_KEY] != digest:
            ret['result'] = False
            ret['comment'] = (
                f'Salt uses the {HASH_METADATA_KEY} metadata key internally,'
                'do not pass it to the boto_s3.object_present state.'
            return ret
    combined_extra_args['Metadata'][HASH_METADATA_KEY] = digest
    # Remove upload-only keys from full set of extra_args
    # to create desired dict for comparisons
    desired_metadata = dict(
        (k, v) for k, v in combined_extra_args.items()  # No need to six anymore
        if k not in UPLOAD_ONLY_EXTRA_ARGS
    )

    # Some args (SSE-C, RequestPayer) must also be passed to get_metadata
    metadata_extra_args = dict(
        (k, v) for k, v in combined_extra_args.items()  # No need for six anymore
        if k in GET_METADATA_EXTRA_ARGS
    )
    r = await hub.exec.boto.s3.get_object_metadata(
        name,
        extra_args=metadata_extra_args,
        region=region,
```

(continues on next page)

(continued from previous page)

```
        key=key,
        keyid=keyid,
        profile=profile,
    )
    if 'error' in r:
        ret['result'] = False
        ret['comment'] = f'Failed to check if S3 object exists: {r["error"]}.' # Use␣
↪fstrings
        return ret

    if r['result']:
        # Check if content and metadata match
        # A hash of the content is injected into the metadata,
        # so we can combine both checks into one
        # Only check metadata keys specified by the user,
        # ignore other fields that have been set
        s3_metadata = dict(
            (k, r['result'][k]) for k in STORED_EXTRA_ARGS
            if k in desired_metadata and k in r['result']
        )
        if s3_metadata == desired_metadata:
            ret['result'] = True
            ret['comment'] = f'S3 object {name} is present.'
            return ret
        action = 'update'
    else:
        s3_metadata = None
        action = 'create'

    # Some Salt code goes out of its way to use salt libs, often it
    # is more appropriate to just call the supporting lib directly
    changes_diff = ''.join(difflib.unified_diff(
        yaml.dump(s3_metadata, default_flow_style=False).splitlines(True),
        yaml.dump(desired_metadata, default_flow_style=False).splitlines(True),
    ))

    if ctx['test']:
        ret['result'] = None
        ret['comment'] = f'S3 object {name} set to be {action}d.'
        ret['comment'] += f'\nChanges:\n{changes_diff}'
        ret['changes'] = {'diff': changes_diff}
        return ret

    r = await hub.boto.s3.upload_file(
        source,
        name,
        extra_args=combined_extra_args,
        region=region,
        key=key,
        keyid=keyid,
        profile=profile,
    )
```

(continues on next page)

(continued from previous page)

```
    if 'error' in r:
        ret['result'] = False
        ret['comment'] = f'Failed to {action} S3 object: {r["error"]}.'
        return ret

    ret['result'] = True
    ret['comment'] = f'S3 object {name} {action}d.'
    ret['comment'] += f'\nChanges:\n{changes_diff}'
    ret['changes'] = {'diff': changes_diff}
    return ret
```

CHAPTER

FORTY

RELEASES

40.1 Idem Release 3

This is the initial public release of Idem, the release number 3 was chosen because the Salt State system should be considered version 1, with an internal version 2.

This release introduces Idem to the world, it takes the Salt State system and migrates it to POP. In doing so the Salt State system has been simplified, extended, and revamped to become a standalone language and interface while following the ideals of POP to make it pluggable into other application stacks.

40.1.1 Now Pluggable!

The Salt State system exists as a single large .py file inside of Salt, the compiler and runtime are all inside a couple of classes and the system is tightly coupled with the Salt minion and execution runtime and environment. This also made the Salt state system very static and difficult to extend. For instance, an old saying on the Salt developer team was "How do we create new requisites for Salt? Ask Tom to make it".

My goal in Idem was to make it in such a way that it could be completely decoupled from Salt, modernize the foundation, add asyncio, and make the system easier to extend. Now the render, compile, and runtime have been separated out, the runtime has been completely rewritten and things like requisites can be added as plugins and runtime rules. Idem can also execute multiple runs concurrently within the same process, and can execute states in parallel or serially.

Idem can execute states in an imperative way or in a declarative way using requisites. This gives developers the best of both worlds. The ability to optimize execution for time or for ease of development and debugging.

40.1.2 Runs Standalone!

The Idem command can be executed against a code tree directly just like a programming language. Instead of setting up minions, masters etc, just make a code tree with sls files and run Idem with the sls fils(s) you want to execute.

40.1.3 Code Sources are Pluggable

Instead of tying the runtime statically to grabbing sources via Salt, the sources are now pluggable. This release only has a local filesystem plugin but it will be easy to add code sources that are over network connections. This should make Idem execution function without needing to have any form of code deployment, but that Idem will be able to execute directly from any network source, like http, S3, or git.

40.1.4 Rendering is Separate

The render system in Salt turned out to be a generally useful system with virtually every attempt to read in files with structured data wanting to be processed though the render system. So for Idem the render system has been separated into a standalone project called *rend*. This project is written in POP and can be app-merged into any other POP project (like idem!). This makes the powerful render system from Salt available to other projects. In fact it is already being used bu other projects like *heist*.

40.1.5 Idem is a Language Runtime

One of the main issues with configuration management tools is that we end up needing to re-write the backend components to work in additional languages and interfaces. The goal of Idem is to make this limitation go away! Instead of making yet another language, Idem ingests structured data. This means that any language can be written on top of Idem as an extension to *rend*. So Idem can be seen not as a yaml based language for idempotent management. But instead as assembly code that languages can be built on top of.

I feel that the language war in configuration management is one of the primary limiting factors for the industry, and why we end up producing new languages to solve specific problems. My hope here is that support for all the managed interfaces can be built into Idem and then made available to any app that wants to use them.

40.2 Idem 4 - Beyond Salt

Idem 4 is a monumental release! This marks the first release where major support for an interface has been made available to app-merge into Idem. This release also marks the first major feature additions to Idem beyond the capabilities found in the Salt state system.

40.2.1 Late Rendering With Render Blocks

This release adds the ability to to execute late rendering using a new feature in *rend* called render blocks. This allows for blocks of code to be rendered during the runtime and added to the overall execution of the state. This makes it easy to break apart the execution to be able to take arbitrary data during the run and apply it to the execution.

40.2.2 Transparent Requisites

Transparent requisites is a powerhouse feature! This new capability allows for state plugins to define requisites that will be automatically added into the mix. This makes it possible for the author of a state plugin to define that if a certain state is ever used, Idem will search the runtime to determine if any of the states defined as transparent requisites have been used and apply them with the desired requite.

40.3 Idem 5 - Encrypted Secrets

Idem 5 comes with a much needed addition, the ability to store encrypted data at rest. This addition introduces a new dep and project that is used for the work of encrypted datastore - Takara. Takara is the standalone manager for keeping track of this data at rest, it allows for data to be easily stored in a pluggable and dynamic way. Takara has also been app-merged into Idem, so you can initiate, unseal, and use takara secret data stores from Idem.

40.4 Idem 5.1

This is a bugfix release of Idem. This release fixes a few issues found inside the state runtime.

For details on the repaired issues please see the following issues on Github:

#11 #12 #13 #14

40.5 Idem 6

With great pleasure we are excited to release Idem 6! This release includes a number of major feature enhancements that make the underlying platform and language significantly more powerful.

40.5.1 Mod System

The mod system allows for last minute injection to execute which can modify the data being executed right before it is run. This also comes with the mod_aggregate system, which allows state modules to implement a function in a state plugin called *mod_aggregate* which can be used to modify the execution right before it is called.

40.5.2 Listen

The new *Listen* requisite allows for modifications to a state to be executed after the entire run completes. This requisite is useful for those who want to be able to react to changes in states without modifying the order of execution.

40.5.3 Any and All Requisites

A new system is now in place that allows for requisites to be triggered based either all requisites being met, or just some of the requisites being met. The new plugin subsystem allows for deeply dynamic handling of this intersection in the evaluation of execution.

40.6 Idem 7

Idem 7 introduces two major new features. A revamped, and easier to use cli, and the new *acct* system.

40.6.1 New CLI

The new cli makes calling states and execution execution modules significantly easier. Now you can just can an sls file directly, or use the existing sls tree settings. This makes idem work more like a programming language.

```
idem state cloud.sls

idem exec cmd.run 'ls -l' shell=True
```

This new simplification should make the use of idem much easier!

40.6.2 The Acct system

The new *acct* system allows for account information to be loaded via plugins into idem. It allows for plugins to be provided by cloud providers that load up the api credentials that should be passed through to the cloud providers.

This systems allows for multiple cloud providers, and multiple accounts, to be targeted simultaneously. So a single execution of idem can orchestrate setting up and maintaining resources across multiple clouds and apis.

It can have cloud specific login plugins, or it can run from a single, encrypted file. The single encrypted file can work across multiple cloud providers and does not require cloud specific plugins.

40.7 Idem 7.1

Idem 7.1 extends the capabilities of Idem 7 by allowing the acct system to be used by *idem exec* instead of just *idem state*. This also makes it possible to pass in a *ctx* option to exec functions, like state functions.

40.8 Idem 7.4

Idem 7.4 adds the *tool* sub for dynamically loading helper functions in idem projects.

40.9 Idem 12.0.0

Idem 12.0.0 introduces 2 new features. kwarg credentials for batch runs, and recursive contracts for exec/state returns.

40.9.1 Recursive Contracts for exec/state returns

States

States already implicitly expect a dictionary with specific keys. With a recursive contracts however, malformed state returns are caught as soon as possible. The expected format for a state return is:

```
{
    "name": "<state name>",
    "changes": {
        "<change name>": {
            "old": <The status of the affected resource before the change>,
            "new": <The status of the affected resource after the change>
        },
    }
    "result": True|False,
    "comment": "<A comment about the state run>",
}
```

Exec

Exec modules now give a warning (in a future release this will be a raised exception) if the return isn't formatted properly. The expected format for an exec return is:

```
{
    "result": True|False,
    "ret": <The return date from the exec module run>,
    "comment": <A status code, exception, or other context for the given result>,
}
```

40.9.2 Kwarg Credentials for internal batch runs

idem.state.batch can run multiple states at once from within python code. This is useful if you don't want to run idem states from the cli, but from your own python project. Now batch runs will accept encrypted or unencrypted acct information as kwargs.

First, put your credentials into a yaml file. Check the documentation for the idem provider to see what the available parameters are.

credentials.yaml:

```
acct_provider_name:
    default:
        profile_kwarg1: value1
        profile_kwarg2: value2
    another_profile_name:
        profile_kwarg1: value1
        profile_kwarg2: value2
```

Use the *acct* program to encrypt the credentials

```
acct encrypt credentials.yaml
```

The output of this command will be the fernet algorithm key used to encrypt your credentials.

output:

```
jIgQhT9j9g9-c5yZ47R95f-zpQ_KYzdrtXwXc5R7eKg=
```

Note: Alternatively, if the *ACCT_KEY* environment variable is set with a valid fernet key, it will be used by the *acct encrypt* command to encrypt the file.

After encrypting credentials.yaml, a new file will have been created called "credentials.yaml.fernet". The contents of this file can be passed to a batch run to safely get provider credentials to idem.

```
cat credentials.yaml.fernet
```

output:

```
gAAAAABgo20bb2XCzM6fN82v7zSaDtVPitexSUk7nhO9MfAUQHvr_YKqjlzMC9NFG3IFt-
↪nie7DqFQH9lRPuhdRrLYoojUBILQ==%
```

Note: It is not wise to send encrypted credentials and their decryption key together; use a pre-defined trusted fernet key at both endpoints of your idem app.

idem app example:

```
import asyncio
import os
import pop.hub
import uuid

hub = pop.hub.Hub()
hub.pop.sub.add(dyne_name="idem")
hub.pop.loop.create()
hub.pop.config.load(["idem"], "idem")

states = {"state name": {"test.succeed_without_changes": {"kwarg1": "value1"}}}
unencrypted_profiles = {
    "provider_name": {"yet_another_profile_name": {"kwarg1": "safe value"}}
}
acct_key = os.environ.get("ACCT_KEY", "jIgQhT9j9g9-c5yZ47R95f-zpQ_KYzdrtXwXc5R7eKg=")

hub.pop.Loop.run_until_complete(
    hub.idem.state.batch(
        states=states,
        profiles=unencrypted_profiles,
        encrypted_profiles="gAAAAABgo20bb2XCzM6fN82v7zSaDtVPitexSUk7nhO9MfAUQHvr_
↪YKqjlzMC9NFG3IFt-nie7DqFQH9lRPuhdRrLYoojUBILQ==",
        acct_key=acct_key,
        default_acct_profile="default",
    )
)
```

40.9.3 Get status of internal batch run

Specify the "name" parameter in idem.state.batch to be able to retrieve the status of an internal run with hub.state.status.

```
status = hub.idem.state.status(name)
print(status)
```

stdout:

```
{
    "sls_sources": [
        "json://{'state name':{'test.succeed_without_changes':{'kwarg1':'value1'}}}"
    ],
    "test": False,
    "errors": [],
    "running": {},
    "acct_profile": "default",
    "status": 0,
```

(continues on next page)

(continued from previous page)

```
    "status_name": "FINISHED",
}
```

40.10 Idem 12.0.2

Idem 12.0.2 rearranges the parameters of *hub.idem.ex.ctx()* and parses them slightly differently.

The "path" or reference on the hub to the exec module being called is the first argument. It can now be just the name of the sub under "exec" that is being referenced.

For example:

In previous versions of *idem*, the *acct_profile* keyword needs to be specified with a full path:

```
ctx = hub.idem.ex.ctx(path="exec.my_cloud.*", acct_profile="my_profile")
```

In *idem* versions 12.0.2 forward, the minimum amount of information is also acceptable:

```
ctx = hub.idem.ex.ctx("my_cloud", "my_profile")
```

This can be used to easily get *acct* information for idem exec module calls in state file jinja:

```
{% set ctx = hub.idem.ex.ctx("my_cloud", "my_profile") %}
test_minimal_ctx:
  test.succeed_with_comment:
    - comment: {{ hub.exec.test.ctx(ctx).ret }}
```

40.11 Idem 13.0.0

Idem 13.0.0 introduces the *describe* subcommand to create SLS files.

40.11.1 Describe Subcommand

A new subcommand, "idem describe" will call the "describe" command for the resource associated with the current account. Using the test states built into idem, we can explore this feature. First we will describe the "test" submodule of idem and output the yaml results to a file called "test.sls".

```
$ idem describe test --output=yaml > test.sls
```

The contents of test.sls will be as follows:

```
Description of test.anop:
  test.anop:
  - name: anop
Description of test.configurable_test_state:
  test.configurable_test_state:
  - name: configurable_test_state
  - changes: true
  - result: true
```

(continues on next page)

(continued from previous page)

```
  - comment: ''
Description of test.fail_with_changes:
  test.fail_with_changes:
  - name: fail_with_changes
Description of test.fail_without_changes:
  test.fail_without_changes:
  - name: fail_without_changes
Description of test.mod_watch:
  test.mod_watch:
  - name: mod_watch
Description of test.none_without_changes:
  test.none_without_changes:
  - name: none_without_changes
Description of test.nop:
  test.nop:
  - name: nop
Description of test.succeed_with_changes:
  test.succeed_with_changes:
  - name: succeed_with_changes
Description of test.succeed_with_comment:
  test.succeed_with_comment:
  - name: succeed_with_comment
  - comment: null
Description of test.succeed_without_changes:
  test.succeed_without_changes:
  - name: succeed_without_changes
Description of test.treq:
  test.treq:
  - name: treq
Description of test.update_low:
  test.update_low:
  - name: update_low
```

The output of "describe" represents a completely valid idem sls file. It shows the current status of your idem resource. You can manage your idem resources by modifying this file then running "idem state" on the file.

```
$ idem state test.sls
```

Output

```
--------
      ID: Description of test.anop
Function: test.anop
  Result: True
 Comment: Success!
 Changes:
--------
      ID: Description of test.configurable_test_state
Function: test.configurable_test_state
  Result: True
 Comment:
 Changes: testing:
```

(continues on next page)

(continued from previous page)

```
    ----------
    old:
        Unchanged
    new:
        Something pretended to change
--------
      ID: Description of test.fail_with_changes
Function: test.fail_with_changes
  Result: False
 Comment: Failure!
 Changes: testing:
    ----------
    old:
        Unchanged
    new:
        Something pretended to change
--------
      ID: Description of test.fail_without_changes
Function: test.fail_without_changes
  Result: False
 Comment: Failure!
 Changes:
--------
      ID: Description of test.mod_watch
Function: test.mod_watch
  Result: True
 Comment: Watch ran!
 Changes: watch:
    True
--------
      ID: Description of test.none_without_changes
Function: test.none_without_changes
  Result: None
 Comment: Success!
 Changes:
--------
      ID: Description of test.nop
Function: test.nop
  Result: True
 Comment: Success!
 Changes:
--------
      ID: Description of test.succeed_with_changes
Function: test.succeed_with_changes
  Result: True
 Comment: Success!
 Changes: testing:
    ----------
    old:
        Unchanged
    new:
        Something pretended to change
```

(continues on next page)

(continued from previous page)

```
--------
      ID: Description of test.succeed_with_comment
Function: test.succeed_with_comment
  Result: True
 Comment: None
 Changes:
--------
      ID: Description of test.succeed_without_changes
Function: test.succeed_without_changes
  Result: True
 Comment: Success!
 Changes:
--------
      ID: Description of test.update_low
Function: test.update_low
  Result: True
 Comment: Success!
 Changes:
--------
      ID: king_arthur
Function: test.nop
  Result: True
 Comment: Success!
 Changes:
--------
      ID: Description of test.treq
Function: test.treq
  Result: True
 Comment: Success!
 Changes:
```

40.11.2 Implementing describe functionality

Create a function called "describe" in a *states* plugin. It should output valid input for the *present* function of that plugin as a dictionary

```
async def describe(hub, ctx):
    # This function returns a description of the resource -- one that "idem state" can␣
↪use to
    # to completely recreate the resource with different credentials
    # or manage the resource with the same credentials
    return {
        "Unique State Name": {
            "reference.to.present.function": {
                [{"name": "resource name"}, {"resource_kwarg": "resource_value"}]
            }
        }
    }
```

40.12 Idem 14.0.0

Idem 14.0.0 introduces *auto_state* exec module contracts. Other contracts are also available for states and exec modules as described below.

40.12.1 Auto State

The "auto_state" contract can be implemented for exec modules. It enforces that a get, list, create, update, and delete function exist for the exec module plugin with specific parameters and returns.

Implementing this contract allows idem to dynamically construct a state for the resource.

Here is an example of how to implement the `auto_state` contract in an exec module plugin:

```
# /my_project_root/my_project/exec/my_cloud/my_resource.py
from typing import Any
from typing import Dict

__contracts__ = ["auto_state"]
__func_alias__ = {"list_": "list"}


async def get(hub, ctx, name, **kwargs) -> Dict[str, Any]:
    """
    Create a dict that describes an instance of this resource.
    The values described with "get" should match what is possible to change with the
↪"update" function.
    If the resource does not exist then return an empty dict.
    """
    result = dict(comment="", result=True, ret=None)
    result["ret"] = {}
    return result


async def list_(hub, ctx, **kwargs) -> Dict[str, Any]:
    """
    List the resource, the "ret" value should be a dict whose keys are a unique name for␣
↪each instance of the resource
    The values are a description of the resource instances that matches the output of␣
↪the "get" function
    """
    result = dict(comment="", result=True, ret=None)
    result["ret"] = {"Unique name": {"key": "value"}}
    result["comment"] = f"Created '{name}'"
    return result


async def create(hub, ctx, name, **kwargs) -> Dict[str, Any]:
    """
    Create the named instance of this resource, assume it does not yet exist.
    Any kwargs added to this function should have a default value (use None if␣
↪necessary).
    Additional kwargs will be used to determine how to create valid "present" states for␣
```

(continues on next page)

(continued from previous page)

```
↪this resource using "idem describe"
    """
    result = dict(comment="", result=True, ret=None)
    result["ret"] = "TODO call the create operation for this resource with **kwargs"
    result["comment"] = f"Created '{name}'"
    return result


async def update(hub, ctx, name, **kwargs) -> Dict[str, Any]:
    """
    Update the named instance of this resource, assume it already exists.
      `ctx.before` has the current status of the resource.
      `kwargs` represents the desired state of the resource
    Compare `ctx.before` to the "kwargs" to update the resource to the desired state.
    """
    result = dict(comment="", result=True, ret=None)
    result["ret"] = "TODO call the update operation for this resource with **kwargs"
    result["comment"] = f"Updated '{name}'"
    return result


async def delete(hub, ctx, name, **kwargs) -> Dict[str, Any]:
    """
    Delete the named instance of this resource, assume it already exists
    """
    result = dict(comment="", result=True, ret=None)
    result["ret"] = "TODO call the delete operation for this resource with **kwargs"
    result["comment"] = f"Deleted '{name}'"
    return result
```

40.12.2 Soft Fail

Implementing the "soft_fail" contract for exec or state plugins will catch any thrown errors. The error message will be injected into the "comment" of the state or exec return and the status will be set to "False".

There can be only one "call" contract for any given function on the hub. "soft_fail" will be overridden for this specific function if your exec module implements another `call` contract.

For state modules:

```
__contracts__ = ["soft_fail"]
```

For exec modules:

```
__contracts__ = ["soft_fail"]
```

40.12.3 Returns

Every exec module and state module implicitly implements the "returns" contract recursively.

This contract enforces that the return values of states and exec modules follow a specific pattern.

exec returns

The return from all exec modules should be a dictionary with the keys "result", "comment", and "ret". For example:

```
def exec_module(hub, ctx):
    return {
        # The result of the exec module operation, "True" if the exec module ran␣
↪successfully, else "False"
        "result": True | False,
        # Any comment on the run of this exec module, such as errors or status codes
        "comment": "",
        # Any return value from this exec module
        "ret": object(),
    }
```

state returns

The return from all state modules should be a dictionary with the keys "result", "comment", "name", and "changes". For example:

```
import dict_tools.differ as difftools


def state_module(hub, ctx, name):
    # The status of the resource before the state was applied
    before = {}
    # The status of the resource after the state was applied
    after = {}
    return {
        # The result of state exec module operation, "True" if the state ran␣
↪successfully, else "False"
        "result": True | False,
        # Any comment on the run of this state module, This is used to qualify HOW the␣
↪state succeeded or failed
        "comment": "",
        # The name that was passed as a parameter to this state module
        "name": name,
        "changes": difftools.deep_diff(before, after),
    }
```

40.12.4 Resource

This contract enforces "present", "absent", and "describe" functions in a `states` plugin. This contract implies the "describe" contract.

```
import dict_tools.differ as difftools

__contracts__ = ["resource"]


async def present(hub, ctx, name, **kwargs):
    """
    Check if a resource exists, if it doesn't create it.
    If the resource exists, make sure that it is in the state described by "kwargs"
    """
    before = {}
    after = {}
    return {
        "result": True | False,
        "comment": "",
        "name": name,
        "changes": difftools.deep_diff(before, after),
    }


async def absent(hub, ctx, name, **kwargs):
    """
    Check if a resource exists, if it does, delete it.
    """
    before = {}
    after = {}
    return {
        "result": True | False,
        "comment": "",
        "name": name,
        "changes": difftools.deep_diff(before, after),
    }


async def describe(hub, ctx):
    """
    Create valid present states for every instance of this resource using the given "ctx"
    """
    return {
        "unique_present_state_name": {
            "present.function.ref": [{"present_kwarg": "present_value"}]
        }
    }
```

40.13 Idem 15.0.0

Idem 15.0.0 introduces reconciliation loop plugin. When invoked from the CLI, the reconciliation loop plugin re-applies the state.

The reconciliation is skipped altogether or stops when the state returns '`result=True`' and there are no '`changes`'.

Reconciliation loop also stops if the state results/changes have not changed during the last 3 iterations.

40.13.1 Reconciler Plugin

Reconciler plugin provided by idem is called 'basic'. It includes a static wait time of 3 seconds between iterations. By default the 'none' plugin is used, which skips reconciliation.

40.13.2 CLI

The reconciler plugin is specified as an argument to the `idem state` CLI.

For example:

```
--reconciler=basic | -R=basic | -r=basic
```

40.13.3 LOOP

To implement a reconcile plugin provide a method like this:

```
async def loop(
    hub,
    name,
    sls_sources,
    render,
    runtime,
    cache_dir,
    sls,
    test,
    acct_file,
    acct_key,
    acct_profile,
):
```

The reconciler loop should return a dictionary like that:

```
{
    "re_runs_count": <number reconciliation loop iterations>,
    "require_re_run": <True or False>,
}
```

40.14 Idem 15.0.1

Idem 15.0.1 introduces 'old_state' and 'new_state' to state returns for the 'resource' contract. It is no longer required to return 'changes' to implement 'resource' contract for Idem states. Idem generates 'changes' automatically based on 'old_state' and 'new_state' in state returns.

Note: It will be required to return 'old_state' and 'new_state' for 'resource' contract in the future release of Idem.

40.15 Idem 16.0.0

Idem 16.0.0 implements evbus in idem cli

40.15.1 Writing an ingress plugin

Refer to the pop-evbus repository for how to write an ingress plugin

40.15.2 Setting up credentials

Configured profiles are formatted as follows:

```
provider:
  profile_name:
      profile_data:
```

The *profile* parameter for the idem event *put* and *put_nowait* functions specifies which profile_name should be used for firing an event. If no profile is specified, the profiles called "default" will be used. There can be multiple providers with the same profile name, the event will be propagated to all providers that have a matching profile name. A context (ctx) will be generated that will be sent to the appropriate ingress plugin's publish function based on *profile*.

Firing Events

40.15.3 from code

The body is any serializable data that comprises the main part of the event The profile is the ingress profile from acct that this event should be published to.

Asynchronous put:

```
async def my_func(hub):
    await hub.idem.event.put(body={"message": "event content"}, profile="default")
```

Synchronous put:

```
def my_func(hub):
    hub.idem.event.put_nowait(body={"message": "event content"}, profile="default")
```

40.15.4 from jinja/sls

Events can also be fired from within an idem sls file via jinja:

```
{%- hub.idem.event.put_nowait(body={"message": "event content"}, profile="default") %}
```

CLI

You can fire one-off events from the CLI like so:

```
idem exec test.event ingress_profile="default" body="my_event" --serialize-plugin="json"
```

Testing

Create a credentials.yml file for connecting to local kafka/rabbitmq containers:

```
pika:
  test_development_evbus_pika:
    connection:
      host: localhost
      port: 5672
      login: guest
      password: guest
    routing_key: my_test_routing_key
kafka:
  test_development_evbus_kafka:
    connection:
      bootstrap_servers: localhost:9092
    topics:
      - my_test_topic
```

Encrypt the credentials file and set the ACCT environment variables

```
$ export ACCT_KEY=$(idem encrypt credentials.yml)
$ export ACCT_FILE="$PWD/credentials.yml.fernet"
```

Start a local rabbitmq server to run the tests:

```
$ docker run -p 5672:5672 \
  --env RABBITMQ_HOSTS=rabbitmq \
  --env RABBITMQ_PORT=5672 \
  --env RABBITMQ_USER=guest \
  --env RABBITMQ_PASS=guest \
  --env RABBITMQ_PROTOCOL=amqp \
  rabbitmq:management
```

Start a local kafka server to run the tests:

```
$ docker run -p 2181:2181 -p 443:9092 -p 9092:9092 \
  --env ADVERTISED_LISTENERS=PLAINTEXT://localhost:443,INTERNAL://localhost:9093 \
  --env LISTENERS=PLAINTEXT://0.0.0.0:9092,INTERNAL://0.0.0.0:9093 \
  --env SECURITY_PROTOCOL_MAP=PLAINTEXT:PLAINTEXT,INTERNAL:PLAINTEXT \
```

(continues on next page)

(continued from previous page)

```
  --env INTER_BROKER=INTERNAL \
  krisgeus/docker-kafka
```

Install the idem test requirements:

```
$ pip install -r requirements/test.in
```

Run the tests with pytest:

```
$ pytest tests
```

Logging Handler

idem adds support for a new logging handler, the queue. All log messages will be published as events to ingress queues with a `idem-logger` profile.

```
idem exec test.event ingress_profile="idem-logger" body="my_event" --log-level=debug --
↪log-handler=event
```

40.16 Idem 17.0.0

Idem 17.0.0 adds support for argument binding to Idem SLS structure. Argument binding references are used to set argument value of a state definition to the result of another state execution.

40.17 Argument Binding References

An argument binding reference sets the state definition argument value to the result of another state execution. In this way, argument binding references determine the order of state execution in the structured layer state (SLS) file structure.

An argument binding reference uses the following format:

> "${<cloud>:<state>:<property_path>}"

Where <cloud> is the state cloud path reference (excluding function reference), <state> is the state declaration ID, and <property_path> is a colon (:) delimited path to the property value.

In the following example, State_B will be executed before State_A because the State_A argument "state_B_id" requires the "ID" value from State_B output.

```
State_A:
   cloud.instance.present:
     - name: "Instance A"
     - state_B_id: "${cloud:State_B:ID}"
 State_B:
   cloud.instance.present:
     - name: "Instance B"
```

40.17.1 Indexes

An argument binding reference can contain an index to point to a specific element of a collection property, as shown in the following example.

```
State_A:
  cloud.instance.present:
    - name: "Instance A"
    - state_B_address: "${cloud:State_B:nics[0]:address}"
State_B:
  cloud.instance.present:
    - name: "Instance B"
    - nics:
        - network_name: "Network_1"
          # address is populated after state is executed
          address:
        - network_name: "Network_2"
          # address is populated after state is executed
          address:
```

An argument binding reference can contain a wildcard (*) index to collect all elements in a collection property. In the following example, State_A "state_B_addresses" argument will be set to a list of 2 addresses, one address for each nic of State_B.

```
State_A:
  cloud.instance.present:
    - name: "Instance A"
    - state_B_addresses: "${cloud:State_B:nics[*]:address}"
State_B:
  cloud.instance.present:
    - name: "Instance B"
    - nics:
        - network_name: "Network_1"
          # address is populated after state is executed
          address:
        - network_name: "Network_2"
          # address is populated after state is executed
          address:
```

40.17.2 "Resource" Contract

To support argument binding, a cloud plugin must implement a "resource" contract, where every state execution function must return a "new_state" property as part of the return dictionary. The "new_state" is used to resolve argument binding requisites.

40.17.3 Arg_bind Requisites

Behind-the-scenes argument binding references are implemented using the Idem requisite system, where argument binding references are parsed during the SLS compilation phase and added to high data as arg_bind requisites. During arg_bind requisite execution, the "new_state" property returned after function execution is used to resolve the value of the referenced parameter.

The following example demonstrates SLS high data after the compilation phase, where "${cloud:State_B:ID}" is resolved as the arg_bind requisite.

```
State_A:
   cloud.instance.present:
     - name: "Instance A"
     - state_B_id: "${cloud:State_B:ID}"
     - arg_bind:
       - cloud:
          - State_B
             - ID: state_B_id
 State_B:
   cloud.instance.present:
     - name: "Instance B"
```

CHAPTER

FORTYONE

CONTRIBUTING GUIDE

Contributions are what make the open source community such an amazing place to learn, inspire, and create in. Any contributions you make are **greatly appreciated!**

41.1 TL;DR Quickstart

1. Have pre-requisites completed:
   - `git`
   - `nox`
   - `pre-commit`
   - Python 3.6+
2. Fork the project
3. `git clone` your fork locally
4. Create your feature branch (ex. `git checkout -b amazing-feature`)
5. Setup your local development environment

```
# setup venv
python3 -m venv .venv
source .venv/bin/activate
pip install -U pip setuptools wheel pre-commit nox

# pre-commit configuration
pre-commit install
```

6. Hack away!
7. Commit your changes (ex. `git commit -m 'Add some amazing-feature'`)
8. Push to the branch (ex. `git push origin amazing-feature`)
9. Open a pull request

For the full details, see below.

41.2 Ways to contribute

We value all contributions, not just contributions to the code. In addition to contributing to the code, you can help the project by:

- Writing, reviewing, and revising documentation, modules, and tutorials
- Opening issues on bugs, feature requests, or docs
- Spreading the word about how great this project is

The rest of this guide will explain our toolchain and how to set up your environment to contribute to the project.

41.3 Overview of how to contribute to this repository

To contribute to this repository, you first need to set up your own local repository:

- *Fork, clone, and branch the repo*
- *Set up your local preview environment*

After this initial setup, you then need to:

- *Sync local master branch with upstream master*
- Edit the documentation in reStructured Text
- *Preview HTML changes locally*
- Open a PR

Once a merge request gets approved, it can be merged!

41.4 Prerequisites

For local development, the following prerequisites are needed:

- git
- Python 3.6+
- Ability to create python venv

41.4.1 Windows 10 users

For the best experience, when contributing from a Windows OS to projects using Python-based tools like `pre-commit`, we recommend setting up Windows Subsystem for Linux (WSL), with the latest version being WSLv2.

The following gists on GitHub have been consulted with success for several contributors:

- Official Microsoft docs on installing WSL
- A list of PowerShell commands in a gist to Enable WSL and Install Ubuntu 20.04
  - Ensure you also read the comment thread below the main content for additional guidance about using Python on the WSL instance.

We recommend Installing Chocolatey on Windows 10 via PowerShell w/ Some Starter Packages. This installs `git`, `microsoft-windows-terminal`, and other helpful tools via the awesome Windows package management tool, Chocolatey.

`choco install git` easily installs `git` for a good Windows-dev experience. From the `git` package page on Chocolatey, the following are installed:

- Git BASH
- Git GUI
- Shell Integration

41.5 Fork, clone, and branch the repo

This project uses the fork and branch Git workflow. For an overview of this method, see Using the Fork-and-Branch Git Workflow.

- First, create a new fork into your personal user space.
- Then, clone the forked repo to your local machine.

```
# SSH or HTTPS
git clone <forked-repo-path>/idem.git
```

Note: Before cloning your forked repo when using SSH, you need to create an SSH key so that your local Git repository can authenticate to the GitLab remote server. See GitLab and SSH keys for instructions, or Connecting to GitHub with SSH.

Configure the remotes for your main upstream repository:

```
# Move into cloned repo
cd idem

# Choose SSH or HTTPS upstream endpoint
git remote add upstream git-or-https-repo-you-forked-from
```

Create new branch for changes to submit:

```
git checkout -b amazing-feature
```

41.6 Set up your local preview environment

If you are not on a Linux machine, you need to set up a virtual environment to preview your local changes and ensure the *prerequisites* are met for a Python virtual environment.

From within your local copy of the forked repo:

```
# Setup venv
python3 -m venv .venv
# If Python 3.6+ is in path as 'python', use the following instead:
# python -m venv .venv
```

(continues on next page)

(continued from previous page)

```
# Activate venv
source .venv/bin/activate
# On Windows, use instead:
# .venv/Scripts/activate

# Install required python packages to venv
pip install -U pip setuptools wheel pre-commit nox
pip install -r requirements/base.txt

# Setup pre-commit
pre-commit install
```

41.6.1 `pre-commit` and `nox` Setup

This project uses pre-commit and nox to make it easier for contributors to get quick feedback, for quality control, and to increase the chance that your merge request will get reviewed and merged.

`nox` handles Sphinx requirements and plugins for you, always ensuring your local packages are the needed versions when building docs. You can think of it as `Make` with superpowers.

41.6.2 What is pre-commit?

`pre-commit` is a tool that will automatically run local tests when you attempt to make a git commit. To view what tests are run, you can view the `.pre-commit-config.yaml` file at the root of the repository.

One big benefit of pre-commit is that *auto-corrective measures* can be done to files that have been updated. This includes Python formatting best practices, proper file line-endings (which can be a problem with repository contributors using differing operating systems), and more.

If an error is found that cannot be automatically fixed, error output will help point you to where an issue may exist.

41.7 Sync local master branch with upstream master

If needing to sync feature branch with changes from upstream master, do the following:

Note: This will need to be done in case merge conflicts need to be resolved locally before a merge to master in the upstream repo.

```
git checkout master
git fetch upstream
git pull upstream master
git push origin master
git checkout my-new-feature
git merge master
```

41.8 Preview HTML changes locally

To ensure that the changes you are implementing are formatted correctly, you should preview a local build of your changes first. To preview the changes:

```
# Activate venv
source .venv/bin/activate
# On Windows, use instead:
# .venv/Scripts/activate

# Generate HTML documentation with nox
nox -e 'docs-html(clean=False)'

# Sphinx website documentation is dumped to docs/_build/html/*
# You can view this locally
# firefox example
firefox docs/_build/html/index.html
```

Note: If you encounter an error, Sphinx may be pointing out formatting errors that need to be resolved in order for `nox` to properly generate the docs.

41.9 Testing a `pop` project

```
# View all nox targets
nox -l

# Output version of Python activated/available
# python --version OR
python3 --version

# Run appropriate test
# Ex. if Python 3.8.x
nox -e 'tests-3.8'
```

This project is a `pop` project which makes use of `pytest-pop`, a `pytest` plugin. For more information on `pytest-pop`, and writing tests for `pop` projects:

- pytest-pop README
- pytest documentation

41.10 Contribution Guidelines

Before asking for a final review for a PR into an idem project, the following guidelines must be met:

41.10.1 Tests

- Tests are written for changes
- Tests provide full coverage of the changed code

41.10.2 Documentation

- Docs are written for feature changes
- Functions have Typehinted parameters
- Code is sufficiently documented with comments
- Parameters are explained in detail in function docstrings
- rst-style examples of the function's usage are included in its docstring

41.10.3 Code Style

- Code is readable and contains comments
- Code contains sufficient logging, including debug logging
- Errors are descriptive
- Follow POP best practices - Plugins are used instead of Classes wherever possible - POP code is accessed via the hub, not python import - Output with hub.log.debug() not print() - The hub is not explicitly passed to functions - Code is organized in the filesystem in a meaningful way - There are no long files – code is separated into plugins with meaningful names (no massive "utils.py" file) - Plugins are organized in a way that will be easily merged with other projects - Subsystems have contracts to standardize plugin structure - *ctx.test* must be implemented in idem states
- Code is written in a re-usable way

41.10.4 Issues

When reporting a bug, the following criteria should be met:

- Bugs include complete steps to reproduce including
  - Bugs include a version report from `pip freeze`
  - Bugs include the full cli command used to reach the error
  - Bugs include sanitized supporting sls/credential files
  - Bugs include output with `--log-level=debug` logging
  - Bugs include the full error output

41.10.5 Pull Requests

- All TODOs are resolved
- All comments by maintainers in code-review are marked “resolved” by maintainers
- All existing tests are passing in the PR pipeline
- The origin pipeline has all tests enabled
- The origin pipeline is visible to maintainers

41.10.6 Versioning

- Backwards-incompatible changes get a major version bump
- New features get a minor version bump
- Bugfixes get a point version bump

CHAPTER

FORTYTWO

LICENSE

Note: For a simplified breakdown of license information, it may be helpful to use tl;drLegal.

```
                                 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
    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.
```

(continues on next page)

(continued from previous page)

```
   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.

   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
```

(continues on next page)

(continued from previous page)

```
   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.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

   To apply the Apache License to your work, attach the following
   boilerplate notice, with the fields enclosed by brackets "[]"
   replaced with your own identifying information. (Don't include
   the brackets!)  The text should be enclosed in the appropriate
   comment syntax for the file format. We also recommend that a
   file or class name and description of purpose be included on the
   same "printed page" as the copyright notice for easier
   identification within third-party archives.

Copyright [2019] [Thomas S Hatch]

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
```

(continues on next page)

(continued from previous page)

```
  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.
```

CHAPTER

FORTYTHREE

INDICES AND TABLES

- genindex
- modindex
- search