stacker Documentation

Release 1.7.2

Michael Barrett

Nov 10, 2020

Contents

stacker is a tool and library used to create & update multiple CloudFormation stacks. It was originally written at Remind and released to the open source community.

stacker Blueprints are written in troposphere, though the purpose of most templates is to keep them as generic as possible and then use configuration to modify them.

At Remind we use stacker to manage all of our Cloudformation stacks - both in development, staging and production without any major issues.

CHAPTER 1

Main Features

- Easily Create/Update/Destroy many stacks in parallel (though with an understanding of cross-stack dependencies)
- Makes it easy to manage large environments in a single config, while still allowing you to break each part of the environment up into its own completely separate stack.
- Manages dependencies between stacks, only launching one after all the stacks it depends on are finished.
- Only updates stacks that have changed and that have not been explicitly locked or disabled.
- Easily pass Outputs from one stack in as Variables on another (which also automatically provides an implicit dependency)
- Use Environments to manage slightly different configuration in different environments.
- Use Lookups to allow dynamic fetching or altering of data used in Variables.
- A diff command for diffing your config against what is running in a live CloudFormation environment.
- A small library of pre-shared Blueprints can be found at the stacker_blueprints repo, making things like setting up a VPC easy.

Contents:

1.1 Organizations using stacker

Below is a list of organizations that currently use stacker in some sense. If you are using stacker, please submit a PR and add your company below!

Remind

> Remind helps educators send quick, simple messages to students and parents on any device. We believe that when communication improves, relationships get stronger. Education gets better.
>
> Remind is the original author of stacker, and has been using it to manage the infrastructure in multiple environments (including production) since early 2015.

Onica

Onica is a global technology consulting company at the forefront of cloud computing. Through collaboration with Amazon Web Services, we help customers embrace a broad spectrum of innovative solutions. From migration strategy to operational excellence, cloud native development, and immersive transformation. Onica is a full spectrum AWS integrator.

AltoStack

AltoStack is a technology and services consultancy specialising in Cloud Consultancy, DevOps, Continuous Delivery and Configuration Management.

From strategy and operations to culture and technology, AltoStack helps businesses identify and address opportunities for growth and profitability.

We are an Amazon Web Services - (AWS) APN Consulting Partner.

Cobli

Cobli develops cutting-edge solutions for fleet management efficiency and intelligence in South America. We bring advanced tracking, analysis and predictions to fleets of any size by connecting vehicles to an easy to use platform through smart devices.

Cobli manages most of its AWS infrastructure using stacker, and we encourage our developers to contribute to free-software whenever possible.

1.2 Terminology

1.2.1 blueprint

A python class that is responsible for creating a CloudFormation template. Usually this is built using troposphere.

1.2.2 config

A YAML config file that defines the *stack definitions* for all of the stacks you want stacker to manage.

1.2.3 environment

A set of variables that can be used inside the config, allowing you to slightly adjust configs based on which environment you are launching.

1.2.4 namespace

A way to uniquely identify a stack. Used to determine the naming of many things, such as the S3 bucket where compiled templates are stored, as well as the prefix for stack names.

1.2.5 stack definition

Defines the *stack* you want to build, usually there are multiple of these in the *config*. It also defines the *variables* to be used when building the *stack*.

1.2.6 stack

The resulting stack of resources that is created by CloudFormation when it executes a template. Each stack managed by stacker is defined by a *stack definition* in the *config*.

1.2.7 output

A CloudFormation Template concept. Stacks can output values, allowing easy access to those values. Often used to export the unique ID's of resources that templates create. Stacker makes it simple to pull outputs from one stack and then use them as a *variable* in another stack.

1.2.8 variable

Dynamic variables that are passed into stacks when they are being built. Variables are defined within the *config*.

1.2.9 lookup

A method for expanding values in the *config* at build time. By default lookups are used to reference Output values from other *stacks* within the same *namespace*.

1.2.10 provider

Provider that supports provisioning rendered *blueprints*. By default, an AWS provider is used.

1.2.11 context

Context is responsible for translating the values passed in via the command line and specified in the *config* to *stacks*.

1.3 Configuration

stacker makes use of a YAML formatted config file to define the different CloudFormation stacks that make up a given environment.

The configuration file has a loose definition, with only a few top-level keywords. Other than those keywords, you can define your own top-level keys to make use of other YAML features like anchors & references to avoid duplicating config. (See *YAML anchors & references* for details)

1.3.1 Top Level Keywords

Namespace

You can provide a **namespace** to create all stacks within. The namespace will be used as a prefix for the name of any stack that stacker creates, and makes it unnecessary to specify the fully qualified name of the stack in output lookups.

In addition, this value will be used to create an S3 bucket that stacker will use to upload and store all CloudFormation templates.

In general, this is paired with the concept of Environments to create a namespace per environment:

```
namespace: ${namespace}
```

Namespace Delimiter

By default, stacker will use '-' as a delimiter between your namespace and the declared stack name to build the actual CloudFormation stack name that gets created. Since child resources of your stacks will, by default, use a portion of your stack name in the auto-generated resource names, the first characters of your fully-qualified stack name potentially convey valuable information to someone glancing at resource names. If you prefer to not use a delimiter, you can pass the **namespace_delimiter** top level key word in the config as an empty string.

See the CloudFormation API Reference for allowed stack name characters

S3 Bucket

Stacker, by default, pushes your CloudFormation templates into an S3 bucket and points CloudFormation at the template in that bucket when launching or updating your stacks. By default it uses a bucket named **stacker-${namespace}**, where the namespace is the namespace provided the config.

If you want to change this, provide the **stacker_bucket** top level key word in the config.

The bucket will be created in the same region that the stacks will be launched in. If you want to change this, or if you already have an existing bucket in a different region, you can set the **stacker_bucket_region** to the region where you want to create the bucket.

S3 Bucket location prior to 1.0.4: There was a "bug" early on in stacker that created the s3 bucket in us-east-1, no matter what you specified as your –region. An issue came up leading us to believe this shouldn't be the expected behavior, so we fixed the behavior. If you executed a stacker build prior to V 1.0.4, your bucket for templates would already exist in us-east-1, requiring you to specify the **stacker_bucket_region** top level keyword.

Note: Deprecation of fallback to legacy template bucket. We will first try the region you defined using the top level keyword under **stacker_bucket_region**, or what was specified in the –region flag. If that fails, we fallback to the us-east-1 region. The fallback to us-east-1 will be removed in a future release resulting in the following botocore excpetion to be thrown:

```
TemplateURL must reference a valid S3 object to which you have access.
```

To avoid this issue, specify the stacker_bucket_region top level keyword as described above. You can specify this keyword now to remove the deprecation warning.

If you want stacker to upload templates directly to CloudFormation, instead of first uploading to S3, you can set **stacker_bucket** to an empty string. However, note that template size is greatly limited when uploading directly. See the CloudFormation Limits Reference.

Module Paths

When setting the `classpath` for blueprints/hooks, it is sometimes desirable to load modules from outside the default `sys.path` (e.g., to include modules inside the same repo as config files).

Adding a path (e.g. `./`) to the **sys_path** top level key word will allow modules from that path location to be used.

Service Role

By default stacker doesn't specify a service role when executing changes to CloudFormation stacks. If you would prefer that it do so, you can set **service_role** to be the ARN of the service that stacker should use when executing CloudFormation changes.

This is the equivalent of setting `RoleARN` on a call to the following CloudFormation api calls: `CreateStack`, `UpdateStack`, `CreateChangeSet`.

See the AWS documentation for AWS CloudFormation Service Roles.

Remote Packages

The **package_sources** top level keyword can be used to define remote sources for blueprints (e.g., retrieving `stacker_blueprints` on github at tag `v1.0.2`).

The only required key for a git repository config is `uri`, but `branch`, `tag`, & `commit` can also be specified:

```
package_sources:
  git:
    - uri: git@github.com:acmecorp/stacker_blueprints.git
    - uri: git@github.com:remind101/stacker_blueprints.git
      tag: 1.0.0
      paths:
        - stacker_blueprints
    - uri: git@github.com:contoso/webapp.git
      branch: staging
    - uri: git@github.com:contoso/foo.git
      commit: 12345678
```

If no specific commit or tag is specified for a repo, the remote repository will be checked for newer commits on every execution of Stacker.

For `.tar.gz` & `zip` archives on s3, specify a `bucket` & `key`:

```
package_sources:
  s3:
    - bucket: mystackers3bucket
      key: archives/blueprints-v1.zip
      paths:
        - stacker_blueprints
    - bucket: anothers3bucket
      key: public/public-blueprints-v2.tar.gz
      requester_pays: true
    - bucket: yetanothers3bucket
      key: sallys-blueprints-v1.tar.gz
      # use_latest defaults to true - will update local copy if the
      # last modified date on S3 changes
      use_latest: false
```

Local directories can also be specified:

```
package_sources:
  local:
    - source: ../vpc
```

Use the `paths` option when subdirectories of the repo/archive/directory should be added to Stacker's `sys.path`.

Cloned repos/archives will be cached between builds; the cache location defaults to ~/.stacker but can be manually specified via the **stacker_cache_dir** top level keyword.

Remote Configs

Configuration yamls from remote configs can also be used by specifying a list of `configs` in the repo to use:

```
package_sources:
  git:
    - uri: git@github.com:acmecorp/stacker_blueprints.git
      configs:
        - vpc.yaml
```

In this example, the configuration in `vpc.yaml` will be merged into the running current configuration, with the current configuration's values taking priority over the values in `vpc.yaml`.

Dictionary Stack Names & Hook Paths

To allow remote configs to be selectively overriden, stack names & hook paths can optionally be defined as dictionaries, e.g.:

```
pre_build:
  my_route53_hook:
    path: stacker.hooks.route53.create_domain:
    required: true
    enabled: true
    args:
      domain: mydomain.com
stacks:
  vpc-example:
    class_path: stacker_blueprints.vpc.VPC
    locked: false
    enabled: true
  bastion-example:
    class_path: stacker_blueprints.bastion.Bastion
    locked: false
    enabled: true
```

Pre & Post Hooks

Many actions allow for pre & post hooks. These are python methods that are executed before, and after the action is taken for the entire config. Hooks can be enabled or disabled, per hook. Only the following actions allow pre/post hooks:

- build (keywords: *pre_build*, *post_build*)
- destroy (keywords: *pre_destroy*, *post_destroy*)

There are a few reasons to use these, though the most common is if you want better control over the naming of a resource than what CloudFormation allows.

The keyword is a list of dictionaries with the following keys:

path: the python import path to the hook

data_key: If set, and the hook returns data (a dictionary), the results will be stored in the context.hook_data with the data_key as its key.

required: whether to stop execution if the hook fails

enabled: whether to execute the hook every stacker run. Default: True. This is a bool that grants you the ability to execute a hook per environment when combined with a variable pulled from an environment file.

args: a dictionary of arguments to pass to the hook

An example using the *create_domain* hook for creating a route53 domain before the build action:

```
pre_build:
  - path: stacker.hooks.route53.create_domain
    required: true
    enabled: true
    args:
      domain: mydomain.com
```

An example of a hook using the `create_domain_bool` variable from the environment file to determine if hook should run. Set `create_domain_bool: true` or `create_domain_bool: false` in the environment file to determine if the hook should run in the environment stacker is running against:

```
pre_build:
  - path: stacker.hooks.route53.create_domain
    required: true
    enabled: ${create_domain_bool}
    args:
      domain: mydomain.com
```

Tags

CloudFormation supports arbitrary key-value pair tags. All stack-level, including automatically created tags, are propagated to resources that AWS CloudFormation supports. See AWS CloudFormation Resource Tags Type for more details. If no tags are specified, the *stacker_namespace* tag is applied to your stack with the value of *namespace* as the tag value.

If you prefer to apply a custom set of tags, specify the top-level keyword *tags* as a map. Example:

```
tags:
  "hello": world
  "my_tag:with_colons_in_key": ${dynamic_tag_value_from_my_env}
  simple_tag: simple value
```

If you prefer to have no tags applied to your stacks (versus the default tags that stacker applies), specify an empty map for the top-level keyword:

```
tags: {}
```

Mappings

Mappings are dictionaries that are provided as Mappings to each CloudFormation stack that stacker produces.

These can be useful for providing things like different AMIs for different instance types in different regions:

```
mappings:
  AmiMap:
    us-east-1:
      NAT: ami-ad227cc4
      ubuntu1404: ami-74e27e1c
      bastion: ami-74e27e1c
    us-west-2:
      NAT: ami-290f4119
      ubuntu1404: ami-5189a661
      bastion: ami-5189a661
```

These can be used in each blueprint/stack as usual.

Lookups

Lookups allow you to create custom methods which take a value and are resolved at build time. The resolved values are passed to the Blueprints before it is rendered. For more information, see the Lookups documentation.

stacker provides some common lookups, but it is sometimes useful to have your own custom lookup that doesn't get shipped with stacker. You can register your own lookups by defining a *lookups* key:

```
lookups:
  custom: path.to.lookup.handler
```

The key name for the lookup will be used as the type name when registering the lookup. The value should be the path to a valid lookup handler.

You can then use these within your config:

```
conf_value: ${custom some-input-here}
```

Stacks

This is the core part of the config - this is where you define each of the stacks that will be deployed in the environment. The top level keyword *stacks* is populated with a list of dictionaries, each representing a single stack to be built.

A stack has the following keys:

name: The logical name for this stack, which can be used in conjuction with the `output` lookup. The value here must be unique within the config. If no `stack_name` is provided, the value here will be used for the name of the CloudFormation stack.

class_path: The python class path to the Blueprint to be used. Specify this or `template_path` for the stack.

template_path: Path to raw CloudFormation template (JSON or YAML). Specify this or `class_path` for the stack. Path can be specified relative to the current working directory (e.g. templates stored alongside the Config), or relative to a directory in the python `sys.path` (i.e. for loading templates retrieved via `packages_sources`).

description: A short description to apply to the stack. This overwrites any description provided in the Blueprint. See: http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/template-description-structure.html

variables: A dictionary of *Variables* to pass into the Blueprint when rendering the CloudFormation template. *Variables* can be any valid YAML data structure.

locked: (optional) If set to true, the stack is locked and will not be updated unless the stack is passed to stacker via the *–force* flag. This is useful for *risky* stacks that you don't want to take the risk of allowing CloudFormation to update, but still want to make sure get launched when the environment is first created. When `locked`, it's not necessary to specify a `class_path` or `template_path`.

enabled: (optional) If set to false, the stack is disabled, and will not be built or updated. This can allow you to disable stacks in different environments.

protected: (optional) When running an update in non-interactive mode, if a stack has *protected* set to *true* and would get changed, stacker will switch to interactive mode for that stack, allowing you to approve/skip the change.

requires: (optional) a list of other stacks this stack requires. This is for explicit dependencies - you do not need to set this if you refer to another stack in a Parameter, so this is rarely necessary.

required_by: (optional) a list of other stacks or targets that require this stack. It's an inverse to `requires`.

tags: (optional) a dictionary of CloudFormation tags to apply to this stack. This will be combined with the global tags, but these tags will take precendence.

stack_name: (optional) If provided, this will be used as the name of the CloudFormation stack. Unlike `name`, the value doesn't need to be unique within the config, since you could have multiple stacks with the same name, but in different regions or accounts. (note: the namespace from the environment will be prepended to this)

region: (optional): If provided, specifies the name of the region that the CloudFormation stack should reside in. If not provided, the default region will be used (`AWS_DEFAULT_REGION`, `~/.aws/config` or the `--region` flag). If both `region` and `profile` are specified, the value here takes precedence over the value in the profile.

profile: (optional): If provided, specifies the name of a AWS profile to use when performing AWS API calls for this stack. This can be used to provision stacks in multiple accounts or regions.

stack_policy_path: (optional): If provided, specifies the path to a JSON formatted stack policy that will be applied when the CloudFormation stack is created and updated. You can use stack policies to prevent CloudFormation from making updates to protected resources (e.g. databases). See: https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/protect-stack-resources.html

in_progress_behavior: (optional): If provided, specifies the behavior for when a stack is in *CREATE_IN_PROGRESS* or *UPDATE_IN_PROGRESS*. By default, stacker will raise an exception if the stack is in an *IN_PROGRESS* state. You can set this option to *wait* and stacker will wait for the previous update to complete before attempting to update the stack.

notification_arns: (optional): If provided, accepts a list of None or many AWS SNS Topic ARNs which will be notified of this stack's CloudFormation state changes.

Stacks Example

Here's an example from stacker_blueprints, used to create a VPC:

```
stacks:
  - name: vpc-example
    class_path: stacker_blueprints.vpc.VPC
    locked: false
    enabled: true
    variables:
      InstanceType: t2.small
      SshKeyName: default
      ImageName: NAT
      AZCount: 2
      PublicSubnets:
        - 10.128.0.0/24
        - 10.128.1.0/24
        - 10.128.2.0/24
        - 10.128.3.0/24
      PrivateSubnets:
        - 10.128.8.0/22
```

(continues on next page)

(continued from previous page)

```
        - 10.128.12.0/22
        - 10.128.16.0/22
        - 10.128.20.0/22
      CidrBlock: 10.128.0.0/16
```

Targets

In stacker, **targets** can be used as a lightweight method to group a number of stacks together, as a named "target" in the graph. Internally, this adds a node to the underlying DAG, which can then be used alongside the *–targets* flag. If you're familiar with the concept of "targets" in systemd, the concept is the same.

name: The logical name for this target.

requires: (optional) a list of stacks or other targets this target requires.

required_by: (optional) a list of stacks or other targets that require this target.

Here's an example of a target that will execute all "database" stacks:

```
targets:
  - name: databases

stacks:
  - name: dbA
    class_path: blueprints.DB
    required_by:
      - databases
  - name: dbB
    class_path: blueprints.DB
    required_by:
      - databases
```

Custom Log Formats

By default, stacker uses the following *log_formats*:

```
log_formats:
  info: "[%(asctime)s] %(message)s"
  color: "[%(asctime)s] \033[%(color)sm%(message)s\033[39m"
  debug: "[%(asctime)s] %(levelname)s %(threadName)s %(name)s:%(lineno)d(
↪%(funcName)s): %(message)s"
```

You may optionally provide custom *log_formats*. In this example, we add the environment name to each log line:

```
log_formats:
  info: "[%(asctime)s] ${environment} %(message)s"
  color: "[%(asctime)s] ${environment} \033[%(color)sm%(message)s\033[39m"
```

You may use any of the standard Python logging module format attributes when building your *log_formats*.

1.3.2 Variables

Variables are values that will be passed into a Blueprint before it is rendered. Variables can be any valid YAML data structure and can leverage Lookups to expand values at build time.

The following concepts make working with variables within large templates easier:

YAML anchors & references

If you have a common set of variables that you need to pass around in many places, it can be annoying to have to copy and paste them in multiple places. Instead, using a feature of YAML known as anchors & references, you can define common values in a single place and then refer to them with a simple syntax.

For example, say you pass a common domain name to each of your stacks, each of them taking it as a Variable. Rather than having to enter the domain into each stack (and hopefully not typo'ing any of them) you could do the following:

```
domain_name: &domain mydomain.com
```

Now you have an anchor called **domain** that you can use in place of any value in the config to provide the value **mydomain.com**. You use the anchor with a reference:

```
stacks:
  - name: vpc
    class_path: stacker_blueprints.vpc.VPC
    variables:
      DomainName: *domain
```

Even more powerful is the ability to anchor entire dictionaries, and then reference them in another dictionary, effectively providing it with default values. For example:

```
common_variables: &common_variables
  DomainName: mydomain.com
  InstanceType: m3.medium
  AMI: ami-12345abc
```

Now, rather than having to provide each of those variables to every stack that could use them, you can just do this instead:

```
stacks:
  - name: vpc
    class_path: stacker_blueprints.vpc.VPC
    variables:
      << : *common_variables
      InstanceType: c4.xlarge # override the InstanceType in this stack
```

Using Outputs as Variables

Since stacker encourages the breaking up of your CloudFormation stacks into entirely separate stacks, sometimes you'll need to pass values from one stack to another. The way this is handled in stacker is by having one stack provide Outputs for all the values that another stack may need, and then using those as the inputs for another stack's *Variables*. stacker makes this easier for you by providing a syntax for *Variables* that will cause stacker to automatically look up the values of Outputs from another stack in its config. To do so, use the following format for the Variable on the target stack:

```
MyParameter: ${output OtherStack::OutputName}
```

Since referencing Outputs from stacks is the most common use case, *output* is the default lookup type. For more information see Lookups.

This example is taken from stacker_blueprints example config - when building things inside a VPC, you will need to pass the *VpcId* of the VPC that you want the resources to be located in. If the *vpc* stack provides an Output called *VpcId*, you can reference it easily:

```
domain_name: my_domain &domain

stacks:
  - name: vpc
    class_path: stacker_blueprints.vpc.VPC
    variables:
      DomainName: *domain
  - name: webservers
    class_path: stacker_blueprints.asg.AutoscalingGroup
    variables:
      DomainName: *domain
      VpcId: ${output vpc::VpcId} # gets the VpcId Output from the vpc stack
```

Note: Doing this creates an implicit dependency from the *webservers* stack to the *vpc* stack, which will cause stacker to submit the *vpc* stack, and then wait until it is complete until it submits the *webservers* stack.

Multi Account/Region Provisioning

You can use stacker to manage CloudFormation stacks in multiple accounts and regions, and reference outputs across them.

As an example, let's say you had 3 accounts you wanted to manage:

1) OpsAccount: An AWS account that has IAM users for employees.
2) ProdAccount: An AWS account for a "production" environment.
3) StageAccount: An AWS account for a "staging" environment.

You want employees with IAM user accounts in OpsAccount to be able to assume roles in both the ProdAccount and StageAccount. You can use stacker to easily manage this:

```
stacks:
  # Create some stacks in both the "prod" and "stage" accounts with IAM roles
  # that employees can use.
  - name: prod/roles
    profile: prod
    class_path: blueprints.Roles
  - name: stage/roles
    profile: stage
    class_path: blueprints.Roles

  # Create a stack in the "ops" account and grant each employee access to
  # assume the roles we created above.
  - name: users
    profile: ops
    class_path: blueprints.IAMUsers
    variables:
      Users:
        john-smith:
          Roles:
            - ${output prod/roles::EmployeeRoleARN}
            - ${output stage/roles::EmployeeRoleARN}
```

Note how I was able to reference outputs from stacks in multiple accounts using the *output* plugin!

1.3.3 Environments

A pretty common use case is to have separate environments that you want to look mostly the same, though with some slight modifications. For example, you might want a *production* and a *staging* environment. The production environment likely needs more instances, and often those instances will be of a larger instance type. Environments allow you to use your existing stacker config, but provide different values based on the environment file chosen on the command line. For more information, see the Environments documentation.

1.3.4 Translators

Note: Translators have been deprecated in favor of Lookups and will be removed in a future release.

Translators allow you to create custom methods which take a value, then modify it before passing it on to the stack. Currently this is used to allow you to pass a KMS encrypted string as a Parameter, then have KMS decrypt it before submitting it to CloudFormation. For more information, see the Translators documentation.

1.4 Environments

When running stacker, you can optionally provide an "environment" file. The environment file defines values, which can then be referred to by name from your stack config file. The environment file is interpreted as YAML if it ends in *.yaml* or *.yml*, otherwise it's interpreted as simple key/value pairs.

1.4.1 Key/Value environments

The stacker config file will be interpolated as a string.Template using the key/value pairs from the environment file. The format of the file is a single key/value per line, separated by a colon (**:**), like this:

```
vpcID: vpc-12345678
```

Provided the key/value vpcID above, you will now be able to use this in your configs for the specific environment you are deploying into. They act as keys that can be used in your config file, providing a sort of templating ability. This allows you to change the values of your config based on the environment you are in. For example, if you have a *webserver* stack, and you need to provide it a variable for the instance size it should use, you would have something like this in your config file:

```
stacks:
  - name: webservers
    class_path: stacker_blueprints.asg.AutoscalingGroup
    variables:
      InstanceType: m3.medium
```

But what if you needed more CPU in your production environment, but not in your staging? Without Environments, you'd need a separate config for each. With environments, you can simply define two different environment files with the appropriate *InstanceType* in each, and then use the key in the environment files in your config. For example:

```
# in the file: stage.env
web_instance_type: m3.medium

# in the file: prod.env
web_instance_type: c4.xlarge
```

(continues on next page)

(continued from previous page)

```
# in your config file:
stacks:
  - name: webservers
    class_path: stacker_blueprints.asg.AutoscalingGroup
    variables:
      InstanceType: ${web_instance_type}
```

1.4.2 YAML environments

YAML environments allow for more complex environment configuration rather than simple text substitution, and support YAML features like anchors and references. To build on the example above, let's define a stack that's a little more complex:

```
stacks:
  - name: webservers
    class_path: stacker_blueprints.asg.AutoscalingGroup
    variables:
      InstanceType: ${web_instance_type}
      IngressCIDRsByPort: ${ingress_cidrs_by_port}
```

We've defined a stack which expects a list of ingress CIDR's allowed access to each port. Our environment files would look like this:

```
# in the file: stage.yml
web_instance_type: m3.medium
ingress_cidrs_by_port:
  80:
    - 192.168.1.0/8
  8080:
    - 0.0.0.0/0

# in the file: prod.env
web_instance_type: c4.xlarge
ingress_cidrs_by_port:
  80:
    - 192.168.1.0/8
  443:
    - 10.0.0.0/16
    - 10.1.0.0/16
```

The YAML format allows for specifying lists, maps, and supports all *pyyaml* functionality allowed in *safe_load()* function.

Variable substitution in the YAML case is a bit more complex than in the *string.Template* case. Objects can only be substituted for variables in the case where we perform a full substitution, such as this:

```
vpcID: ${vpc_variable}
```

We can not substitute an object in a sub-string, such as this:

```
vpcID: prefix-${vpc_variable}
```

It makes no sense to substitute a complex object in this case, and we will raise an error if that happens. You can still perform this substitution with primitives; numbers, strings, but not dicts or lists.

Note: Namespace defined in the environment file has been deprecated in favor of defining the namespace in the config and will be removed in a future release.

1.5 Translators

Note: Translators have been deprecated in favor of Lookups and will be removed in a future release.

Stacker provides the ability to dynamically replace values in the config via a concept called translators. A translator is meant to take a value and convert it by calling out to another service or system. This is initially meant to deal with encrypting fields in your config.

Translators are custom YAML constructors. As an example, if you have a database and it has a parameter called `DBPassword` that you don't want to store in clear text in your config (maybe because you want to check it into your version control system to share with the team), you could instead encrypt the value using `kms`. For example:

```
# We use the aws cli to get the encrypted value for the string
# "PASSWORD" using the master key called 'myStackerKey' in us-east-1
$ aws --region us-east-1 kms encrypt --key-id alias/myStackerKey \
    --plaintext "PASSWORD" --output text --query CiphertextBlob

CiD6bC8t2Y<...encrypted blob...>

# In stacker we would reference the encrypted value like:
DBPassword: !kms us-east-1@CiD6bC8t2Y<...encrypted blob...>

# The above would resolve to
DBPassword: PASSWORD
```

This requires that the person using stacker has access to the master key used to encrypt the value.

It is also possible to store the encrypted blob in a file (useful if the value is large) using the *file://* prefix, ie:

```
DockerConfig: !kms file://dockercfg
```

Note: Translators resolve the path specified with *file://* relative to the location of the config file, not where the stacker command is run.

1.6 Lookups

Stacker provides the ability to dynamically replace values in the config via a concept called lookups. A lookup is meant to take a value and convert it by calling out to another service or system.

A lookup is denoted in the config with the `${<lookup type> <lookup input>}` syntax. If `<lookup type>` isn't provided, stacker will fall back to use the `output` lookup .

Lookups are only resolved within Variables. They can be nested in any part of a YAML data structure and within another lookup itself.

Note: If a lookup has a non-string return value, it can be the only lookup within a value.

ie. if *custom* returns a list, this would raise an exception:

```
Variable: ${custom something}, ${output otherStack::Output}
```

This is valid:

```
Variable: ${custom something}
```

For example, given the following:

```
stacks:
  - name: sg
    class_path: some.stack.blueprint.Blueprint
    variables:
      Roles:
        - ${output otherStack::IAMRole}
      Values:
        Env:
          Custom: ${custom ${output otherStack::Output}}
          DBUrl: postgres://${output dbStack::User}@${output dbStack::HostName}
```

The Blueprint would have access to the following resolved variables dictionary:

```
# variables
{
  "Roles": ["other-stack-iam-role"],
  "Values": {
    "Env": {
      "Custom": "custom-output",
      "DBUrl": "postgres://user@hostname",
    },
  },
}
```

stacker includes the following lookup types:

- *output lookup*
- *ami lookup*
- *custom lookup*
- *default lookup*
- *dynamodb lookup*
- *envvar lookup*
- *file lookup*
- *hook_data lookup*
- *kms lookup*
- *rxref lookup*
- *ssmstore lookup*
- *xref lookup*

1.6.1 Output Lookup

The `output` lookup takes a value of the format: `<stack name>::<output name>` and retrieves the output from the given stack name within the current namespace.

stacker treats output lookups differently than other lookups by auto adding the referenced stack in the lookup as a requirement to the stack whose variable the output value is being passed to.

You can specify an output lookup with the following syntax:

```
ConfVariable: ${output someStack::SomeOutput}
```

1.6.2 default Lookup

The `default` lookup type will check if a value exists for the variable in the environment file, then fall back to a default defined in the stacker config if the environment file doesn't contain the variable. This allows defaults to be set at the config file level, while granting the user the ability to override that value per environment.

Format of value:: <env_var>::<default value>

For example:: Groups: ${default app_security_groups::sg-12345,sg-67890}

If *app_security_groups* is defined in the environment file, its defined value will be returned. Otherwise, *sg-12345,sg-67890* will be the returned value.

Note: The `default` lookup only supports checking if a variable is defined in an environment file. It does not support other embedded lookups to see if they exist. Only checking variables in the environment file are supported. If you attempt to have the default lookup perform any other lookup that fails, stacker will throw an exception for that lookup and will stop your build before it gets a chance to fall back to the default in your config.

1.6.3 KMS Lookup

The `kms` lookup type decrypts its input value.

As an example, if you have a database and it has a parameter called `DBPassword` that you don't want to store in clear text in your config (maybe because you want to check it into your version control system to share with the team), you could instead encrypt the value using `kms`.

For example:

```
# We use the aws cli to get the encrypted value for the string
# "PASSWORD" using the master key called 'myStackerKey' in us-east-1
$ aws --region us-east-1 kms encrypt --key-id alias/myStackerKey \
    --plaintext "PASSWORD" --output text --query CiphertextBlob

CiD6bC8t2Y<...encrypted blob...>

# In stacker we would reference the encrypted value like:
DBPassword: ${kms us-east-1@CiD6bC8t2Y<...encrypted blob...>}

# The above would resolve to
DBPassword: PASSWORD
```

This requires that the person using stacker has access to the master key used to encrypt the value.

It is also possible to store the encrypted blob in a file (useful if the value is large) using the `file://` prefix, ie:

```
DockerConfig: ${kms file://dockercfg}
```

Note: Lookups resolve the path specified with *file://* relative to the location of the config file, not where the stacker command is run.

1.6.4 XRef Lookup

The `xref` lookup type is very similar to the `output` lookup type, the difference being that `xref` resolves output values from stacks that aren't contained within the current stacker namespace, but are existing stacks containing outputs within the same region on the AWS account you are deploying into. `xref` allows you to lookup these outputs from the stacks already on your account by specifying the stacks fully qualified name in the CloudFormation console.

Where the `output` type will take a stack name and use the current context to expand the fully qualified stack name based on the namespace, `xref` skips this expansion because it assumes you've provided it with the fully qualified stack name already. This allows you to reference output values from any CloudFormation stack in the same region.

Also, unlike the `output` lookup type, `xref` doesn't impact stack requirements.

For example:

```
ConfVariable: ${xref fully-qualified-stack::SomeOutput}
```

1.6.5 RXRef Lookup

The `rxref` lookup type is very similar to the `xref` lookup type, the difference being that `rxref` will lookup output values from stacks that are relative to the current namespace but external to the stack, but will not resolve them. `rxref` assumes the stack containing the output already exists.

Where the `xref` type assumes you provided a fully qualified stack name, `rxref`, like `output` expands and retrieves the output from the given stack name within the current namespace, even if not defined in the stacker config you provided it.

Because there is no requirement to keep all stacks defined within the same stacker YAML config, you might need the ability to read outputs from other stacks deployed by stacker into your same account under the same namespace. `rxref` gives you that ability. This is useful if you want to break up very large configs into smaller groupings.

Also, unlike the `output` lookup type, `rxref` doesn't impact stack requirements.

For example:

```
# in stacker.env
namespace: MyNamespace

# in stacker.yml
ConfVariable: ${rxref my-stack::SomeOutput}

# the above would effectively resolve to
ConfVariable: ${xref MyNamespace-my-stack::SomeOutput}
```

Although possible, it is not recommended to use `rxref` for stacks defined within the same stacker YAML config.

1.6.6 File Lookup

The `file` lookup type allows the loading of arbitrary data from files on disk. The lookup additionally supports using a `codec` to manipulate or wrap the file contents prior to injecting it. The parameterized-b64 `codec` is particularly useful to allow the interpolation of CloudFormation parameters in a UserData attribute of an instance or launch configuration.

Basic examples:

```
# We've written a file to /some/path:
$ echo "hello there" > /some/path

# In stacker we would reference the contents of this file with the following
conf_key: ${file plain:file://some/path}

# The above would resolve to
conf_key: hello there

# Or, if we used wanted a base64 encoded copy of the file data
conf_key: ${file base64:file://some/path}

# The above would resolve to
conf_key: aGVsbG8gdGhlcmUK
```

Supported codecs:

- plain - load the contents of the file untouched. This is the only codec that should be used with raw Cloudformation templates (the other codecs are intended for blueprints).
- base64 - encode the plain text file at the given path with base64 prior to returning it
- parameterized - the same as plain, but additionally supports referencing CloudFormation parameters to create userdata that's supplemented with information from the template, as is commonly needed in EC2 UserData. For example, given a template parameter of BucketName, the file could contain the following text:

  ```
  #!/bin/sh
  aws s3 sync s3://{{BucketName}}/somepath /somepath
  ```

  and then you could use something like this in the YAML config file:

  ```
  UserData: ${file parameterized:/path/to/file}
  ```

  resulting in the UserData parameter being defined as:

  ```
  { "Fn::Join" : ["", [
    "#!/bin/sh\naws s3 sync s3://",
    {"Ref" : "BucketName"},
    "/somepath /somepath"
  ]] }
  ```

- parameterized-b64 - the same as parameterized, with the results additionally wrapped in { "Fn::Base64": ... } , which is what you actually need for EC2 UserData
- json - decode the file as JSON and return the resulting object
- json-parameterized - Same as `json`, but applying templating rules from `parameterized` to every object *value*. Note that object *keys* are not modified. Example (an external PolicyDocument):

```
{
 "Version": "2012-10-17",
 "Statement": [
   {
     "Effect": "Allow",
     "Action": [
       "some:Action"
     ],
     "Resource": "{{MyResource}}"
   }
 ]
}
```

- yaml - decode the file as YAML and return the resulting object. All strings are returned as `unicode` even in Python 2.
- yaml-parameterized - Same as `json-parameterized`, but using YAML. Example:

```
Version: 2012-10-17
Statement
  - Effect: Allow
    Action:
      - "some:Action"
    Resource: "{{MyResource}}"
```

When using parameterized-b64 for UserData, you should use a local_parameter defined as such:

```
from troposphere import AWSHelperFn

"UserData": {
  "type": AWSHelperFn,
  "description": "Instance user data",
  "default": Ref("AWS::NoValue")
}
```

and then assign UserData in a LaunchConfiguration or Instance to self.get_variables()["UserData"]. Note that we use AWSHelperFn as the type because the parameterized-b64 codec returns either a Base64 or a GenericHelperFn troposphere object.

1.6.7 SSM Parameter Store Lookup

The `ssmstore` lookup type retrieves a value from the Simple Systems Manager Parameter Store.

As an example, if you have a database and it has a parameter called `DBUser` that you don't want to store in clear text in your config, you could instead store it as a SSM parameter named `MyDBUser`.

For example:

```
# We use the aws cli to store the database username
$ aws ssm put-parameter --name "MyDBUser" --type "String" \
    --value "root"

# In stacker we would reference the value like:
DBUser: ${ssmstore us-east-1@MyDBUser}

# Which would resolve to:
DBUser: root
```

Encrypted values ("SecureStrings") can also be used, which will be automatically decrypted (assuming the Stacker user has access to the associated KMS key). Care should be taken when using this with encrypted values (i.e. a safe policy is to only use it with `no_echo` CFNString values)

The region can be omitted (e.g. `DBUser: ${ssmstore MyDBUser}`), in which case `us-east-1` will be assumed.

1.6.8 DynamoDb Lookup

The `dynamodb` lookup type retrieves a value from a DynamoDb table.

As an example, if you have a Dynamo Table named `TestTable` and it has an Item with a Primary Partition key called `TestKey` and a value named `BucketName` , you can look it up by using Stacker. The lookup key in this case is TestVal

For example:

```
# We can reference that dynamo value
BucketName: ${dynamodb us-east-1:TestTable@TestKey:TestVal.BucketName}

# Which would resolve to:
BucketName: stacker-test-bucket
```

You can lookup other data types by putting the data type in the lookup. Valid values are "S"(String), "N"(Number), "M"(Map), "L"(List).

For example:

```
ServerCount: ${dynamodb us-east-1:TestTable@TestKey:TestVal.ServerCount[N]}

This would return an int value, rather than a string
```

You can lookup values inside of a map:

For example:

```
ServerCount: ${dynamodb us-east-1:TestTable@TestKey:TestVal.ServerInfo[M].
                                                          ServerCount[N]}
```

1.6.9 Shell Environment Lookup

The `envvar` lookup type retrieves a value from a variable in the shell's environment.

Example:

```
# Set an environment variable in the current shell.
$ export DATABASE_USER=root

# In the stacker config we could reference the value:
DBUser: ${envvar DATABASE_USER}

# Which would resolve to:
DBUser: root
```

You can also get the variable name from a file, by using the `file://` prefix in the lookup, like so:

```
DBUser: ${envvar file://dbuser_file.txt}
```

1.6.10 EC2 AMI Lookup

The `ami` lookup is meant to search for the most recent AMI created that matches the given filters.

Valid arguments:

```
region OPTIONAL ONCE:
    e.g. us-east-1@

owners (comma delimited) REQUIRED ONCE:
    aws_account_id | amazon | self

name_regex (a regex) REQUIRED ONCE:
    e.g. my-ubuntu-server-[0-9]+

executable_users (comma delimited) OPTIONAL ONCE:
    aws_account_id | amazon | self
```

Any other arguments specified are sent as filters to the aws api For example, "architecture:x86_64" will add a filter.

Example:

```
# Grabs the most recently created AMI that is owned by either this account,
# amazon, or the account id 888888888888 that has a name that matches
# the regex "server[0-9]+" and has "i386" as its architecture.

# Note: The region is optional, and defaults to the current stacker region
ImageId: ${ami [<region>@]owners:self,888888888888,amazon name_regex:server[0-9]+␣
↪architecture:i386}
```

1.6.11 Hook Data Lookup

When using hooks, you can have the hook store results in the hook_data dictionary on the context by setting *data_key* in the hook config.

This lookup lets you look up values in that dictionary. A good example of this is when you use the aws_lambda hook to upload AWS Lambda code, then need to pass that code object as the *Code* variable in the aws_lambda blueprint dictionary.

Example:

```
# If you set the "data_key" config on the aws_lambda hook to be "myfunction"
# and you name the function package "TheCode" you can get the troposphere
# awslambda.Code object with:

Code: ${hook_data myfunction::TheCode}
```

1.6.12 Custom Lookup

A custom lookup may be registered within the config. For more information see Configuring Lookups.

1.7 Commands

1.7.1 Build

Build is used to create/update the stacks provided in the config file. It automatically figures out any dependencies between stacks, and creates them in parallel safely (if a stack depends on another stack, it will wait for that stack to be finished before updating/creating).

It also provides the *–dump* flag for testing out blueprints before pushing them up into CloudFormation. Even then, some errors might only be noticed after first submitting a stack, at which point it can no longer be updated by Stacker. When that situation is detected in interactive mode, you will be prompted to delete and re-create the stack, so that you don't need to do it manually in the AWS console. If that behavior is also desired in non-interactive mode, enable the *–recreate-failed* flag.

```
# stacker build -h
usage: stacker build [-h] [-e ENV=VALUE] [-r REGION] [-v] [-i]
                     [--replacements-only] [--recreate-failed] [-o]
                     [--force STACKNAME] [--stacks STACKNAME] [-t] [-d DUMP]
                     [environment] config

Launches or updates CloudFormation stacks based on the given config. Stacker
is smart enough to figure out if anything (the template or parameters) have
changed for a given stack. If nothing has changed, stacker will correctly skip
executing anything against the stack.

positional arguments:
  environment           Path to a simple `key: value` pair environment file.
                        The values in the environment file can be used in the
                        stack config as if it were a string.Template type:
                        https://docs.python.org/2/library/string.html
                        #template-strings.
  config                The config file where stack configuration is located.
                        Must be in yaml format. If `-` is provided, then the
                        config will be read from stdin.

optional arguments:
  -h, --help            show this help message and exit
  -e ENV=VALUE, --env ENV=VALUE
                        Adds environment key/value pairs from the command
                        line. Overrides your environment file settings. Can be
                        specified more than once.
  -r REGION, --region REGION
                        The AWS region to launch in.
  -v, --verbose         Increase output verbosity. May be specified up to
                        twice.
  -i, --interactive     Enable interactive mode. If specified, this will use
                        the AWS interactive provider, which leverages
                        Cloudformation Change Sets to display changes before
                        running cloudformation templates. You'll be asked if
                        you want to execute each change set. If you only want
                        to authorize replacements, run with "--replacements-
                        only" as well.
  --replacements-only   If interactive mode is enabled, stacker will only
                        prompt to authorize replacements.
  --recreate-failed     Destroy and re-create stacks that are stuck in a
                        failed state from an initial deployment when updating.
```

(continues on next page)

(continued from previous page)

```
  -o, --outline         Print an outline of what steps will be taken to build
                        the stacks
  --force STACKNAME     If a stackname is provided to --force, it will be
                        updated, even if it is locked in the config.
  --stacks STACKNAME    Only work on the stacks given. Can be specified more
                        than once. If not specified then stacker will work on
                        all stacks in the config file.
  -t, --tail            Tail the CloudFormation logs while working with stacks
  -d DUMP, --dump DUMP  Dump the rendered Cloudformation templates to a
                        directory
```

1.7.2 Destroy

Destroy handles the tearing down of CloudFormation stacks defined in the config file. It figures out any dependencies that may exist, and destroys the stacks in the correct order (in parallel if all dependent stacks have already been destroyed).

```
# stacker destroy -h
usage: stacker destroy [-h] [-e ENV=VALUE] [-r REGION] [-v] [-i]
                       [--replacements-only] [-f] [--stacks STACKNAME] [-t]
                       environment config

Destroys CloudFormation stacks based on the given config. Stacker will
determine the order in which stacks should be destroyed based on any manual
requirements they specify or output values they rely on from other stacks.

positional arguments:
  environment           Path to a simple `key: value` pair environment file.
                        The values in the environment file can be used in the
                        stack config as if it were a string.Template type:
                        https://docs.python.org/2/library/string.html
                        #template-strings. Must define at least a "namespace".
  config                The config file where stack configuration is located.
                        Must be in yaml format. If `-` is provided, then the
                        config will be read from stdin.

optional arguments:
  -h, --help            show this help message and exit
  -e ENV=VALUE, --env ENV=VALUE
                        Adds environment key/value pairs from the command
                        line. Overrides your environment file settings. Can be
                        specified more than once.
  -r REGION, --region REGION
                        The AWS region to launch in.
  -v, --verbose         Increase output verbosity. May be specified up to
                        twice.
  -i, --interactive     Enable interactive mode. If specified, this will use
                        the AWS interactive provider, which leverages
                        Cloudformation Change Sets to display changes before
                        running cloudformation templates. You'll be asked if
                        you want to execute each change set. If you only want
                        to authorize replacements, run with "--replacements-
                        only" as well.
  --replacements-only   If interactive mode is enabled, stacker will only
                        prompt to authorize replacements.
```

(continues on next page)

(continued from previous page)

```
  -f, --force           Whether or not you want to go through with destroying
                        the stacks
  --stacks STACKNAME    Only work on the stacks given. Can be specified more
                        than once. If not specified then stacker will work on
                        all stacks in the config file.
  -t, --tail            Tail the CloudFormation logs while working with stacks
```

1.7.3 Info

Info displays information on the CloudFormation stacks based on the given config.

```
# stacker info -h
usage: stacker info [-h] [-e ENV=VALUE] [-r REGION] [-v] [-i]
                    [--replacements-only] [--stacks STACKNAME]
                    environment config

Gets information on the CloudFormation stacks based on the given config.

positional arguments:
  environment           Path to a simple `key: value` pair environment file.
                        The values in the environment file can be used in the
                        stack config as if it were a string.Template type:
                        https://docs.python.org/2/library/string.html
                        #template-strings. Must define at least a "namespace".
  config                The config file where stack configuration is located.
                        Must be in yaml format. If `-` is provided, then the
                        config will be read from stdin.

optional arguments:
  -h, --help            show this help message and exit
  -e ENV=VALUE, --env ENV=VALUE
                        Adds environment key/value pairs from the command
                        line. Overrides your environment file settings. Can be
                        specified more than once.
  -r REGION, --region REGION
                        The AWS region to launch in.
  -v, --verbose         Increase output verbosity. May be specified up to
                        twice.
  -i, --interactive     Enable interactive mode. If specified, this will use
                        the AWS interactive provider, which leverages
                        Cloudformation Change Sets to display changes before
                        running cloudformation templates. You'll be asked if
                        you want to execute each change set. If you only want
                        to authorize replacements, run with "--replacements-
                        only" as well.
  --replacements-only   If interactive mode is enabled, stacker will only
                        prompt to authorize replacements.
  --stacks STACKNAME    Only work on the stacks given. Can be specified more
                        than once. If not specified then stacker will work on
                        all stacks in the config file.
```

1.7.4 Diff

Diff creates a CloudFormation Change Set for each stack and displays the resulting changes. This works for stacks that already exist and new stacks.

For stacks that are dependent on outputs from other stacks in the same file, stacker will infer that an update was made to the "parent" stack and invalidate outputs from resources that were changed and replace their value with `<inferred-change: stackName.outputName=unresolvedValue>`. This is done to illustrate the potential blast radius of a change and assist in tracking down why subsequent stacks could change. This inference is not perfect but takes a "best effort" approach to showing potential change between stacks that rely on each others outputs.

```
# stacker diff -h
usage: stacker diff [-h] [-e ENV=VALUE] [-r REGION] [-v] [-i]
                    [--replacements-only] [--force STACKNAME]
                    [--stacks STACKNAME]
                    environment config

Diffs the config against the currently running CloudFormation stacks Sometimes
small changes can have big impacts. Run "stacker diff" before "stacker build"
to detect bad things(tm) from happening in advance!

positional arguments:
  environment           Path to a simple `key: value` pair environment file.
                        The values in the environment file can be used in the
                        stack config as if it were a string.Template type:
                        https://docs.python.org/2/library/string.html
                        #template-strings. Must define at least a "namespace".
  config                The config file where stack configuration is located.
                        Must be in yaml format. If `-` is provided, then the
                        config will be read from stdin.

optional arguments:
  -h, --help            show this help message and exit
  -e ENV=VALUE, --env ENV=VALUE
                        Adds environment key/value pairs from the command
                        line. Overrides your environment file settings. Can be
                        specified more than once.
  -r REGION, --region REGION
                        The AWS region to launch in.
  -v, --verbose         Increase output verbosity. May be specified up to
                        twice.
  -i, --interactive     Enable interactive mode. If specified, this will use
                        the AWS interactive provider, which leverages
                        Cloudformation Change Sets to display changes before
                        running cloudformation templates. You'll be asked if
                        you want to execute each change set. If you only want
                        to authorize replacements, run with "--replacements-
                        only" as well.
  --replacements-only   If interactive mode is enabled, stacker will only
                        prompt to authorize replacements.
  --force STACKNAME     If a stackname is provided to --force, it will be
                        diffed, even if it is locked in the config.
  --stacks STACKNAME    Only work on the stacks given. Can be specified more
                        than once. If not specified then stacker will work on
                        all stacks in the config file.
```

1.8 Blueprints

Blueprints are python classes that dynamically build CloudFormation templates. Where you would specify a raw Cloudformation template in a stack using the `template_path` key, you instead specify a blueprint python file using the `class_path` key.

Traditionally blueprints are built using troposphere, but that is not absolutely necessary. You are encouraged to check out the library of publicly shared Blueprints in the stacker_blueprints package.

Making your own should be easy, and you can take a lot of examples from stacker_blueprints. In the end, all that is required is that the Blueprint is a subclass of *stacker.blueprints.base* and it have the following methods:

```
# Initializes the blueprint
def __init__(self, name, context, mappings=None):

# Updates self.template to create the actual template
def create_template(self):

# Returns a tuple: (version, rendered_template)
def render_template(self):
```

1.8.1 Variables

A Blueprint can define a `VARIABLES` property that defines the variables it accepts from the Config Variables.

`VARIABLES` should be a dictionary of `<variable name>: <variable definition>`. The variable definition should be a dictionary which supports the following optional keys:

type: The type for the variable value. This can either be a native python type or one of the *Variable Types*.

default: The default value that should be used for the variable if none is provided in the config.

description: A string that describes the purpose of the variable.

validator: An optional function that can do custom validation of the variable. A validator function should take a single argument, the value being validated, and should return the value if validation is successful. If there is an issue validating the value, an exception (`ValueError`, `TypeError`, etc) should be raised by the function.

no_echo: Only valid for variables whose type subclasses `CFNType`. Whether to mask the parameter value whenever anyone makes a call that describes the stack. If you set the value to true, the parameter value is masked with asterisks (*).

allowed_values: Only valid for variables whose type subclasses `CFNType`. The set of values that should be allowed for the CloudFormation Parameter.

allowed_pattern: Only valid for variables whose type subclasses `CFNType`. A regular expression that represents the patterns you want to allow for the CloudFormation Parameter.

max_length: Only valid for variables whose type subclasses `CFNType`. The maximum length of the value for the CloudFormation Parameter.

min_length: Only valid for variables whose type subclasses `CFNType`. The minimum length of the value for the CloudFormation Parameter.

max_value: Only valid for variables whose type subclasses `CFNType`. The max value for the CloudFormation Parameter.

min_value: Only valid for variables whose type subclasses `CFNType`. The min value for the CloudFormation Parameter.

constraint_description: Only valid for variables whose type subclasses `CFNType`. A string that explains the constraint when the constraint is violated for the CloudFormation Parameter.

1.8.2 Variable Types

Any native python type can be specified as the `type` for a variable. You can also use the following custom types:

TroposphereType

The `TroposphereType` can be used to generate resources for use in the blueprint directly from user-specified configuration. Which case applies depends on what `type` was chosen, and how it would be normally used in the blueprint (and CloudFormation in general).

Resource Types

When `type` is a Resource Type, the value specified by the user in the configuration file must be a dictionary, but with two possible structures.

When `many` is disabled, the top-level dictionary keys correspond to parameters of the `type` constructor. The key-value pairs will be used directly, and one object will be created and stored in the variable.

When `many` is enabled, the top-level dictionary *keys* are resource titles, and the corresponding *values* are themselves dictionaries, to be used as parameters for creating each of multiple `type` objects. A list of those objects will be stored in the variable.

Property Types

When `type` is a Property Type the value specified by the user in the configuration file must be a dictionary or a list of dictionaries.

When `many` is disabled, the top-level dictionary keys correspond to parameters of the `type` constructor. The key-value pairs will be used directly, and one object will be created and stored in the variable.

When `many` is enabled, a list of dictionaries is expected. For each element, one corresponding call will be made to the `type` constructor, and all the objects produced will be stored (also as a list) in the variable.

Optional variables

In either case, when `optional` is enabled, the variable may have no value assigned, or be explicitly assigned a null value. When that happens the variable's final value will be `None`.

Example

Below is an annotated example:

```
from stacker.blueprints.base import Blueprint
from stacker.blueprints.variables.types import TroposphereType
from troposphere import s3, sns

class Buckets(Blueprint):
```

(continues on next page)

(continued from previous page)

```
    VARIABLES = {
        # Specify that Buckets will be a list of s3.Bucket types.
        # This means the config should a dictionary of dictionaries
        # which will be converted into troposphere buckets.
        "Buckets": {
            "type": TroposphereType(s3.Bucket, many=True),
            "description": "S3 Buckets to create.",
        },
        # Specify that only a single bucket can be passed.
        "SingleBucket": {
            "type": TroposphereType(s3.Bucket),
            "description": "A single S3 bucket",
        },
        # Specify that Subscriptions will be a list of sns.Subscription types.
        # Note: sns.Subscription is the property type, not the standalone
        # sns.SubscriptionResource.
        "Subscriptions": {
            "type": TroposphereType(sns.Subscription, many=True),
            "description": "Multiple SNS subscription designations"
        },
        # Specify that only a single subscription can be passed, and that it
        # is made optional.
        "SingleOptionalSubscription": {
            "type": TroposphereType(sns.Subscription, optional=True),
            "description": "A single, optional SNS subscription designation"
        }
    }

    def create_template(self):
        t = self.template
        variables = self.get_variables()

        # The Troposphere s3 buckets have already been created when we
        access variables["Buckets"], we just need to add them as
        resources to the template.
        [t.add_resource(bucket) for bucket in variables["Buckets"]]

        # Add the single bucket to the template. You can use
        `Ref(single_bucket)` to pass CloudFormation references to the
        bucket just as you would with any other Troposphere type.
        single_bucket = variables["SingleBucket"]
        t.add_resource(single_bucket)

        subscriptions = variables["Subscriptions"]
        optional_subscription = variables["SingleOptionalSubscription"]
        # Handle it in some special way...
        if optional_subscription is not None:
            subscriptions.append(optional_subscription)

        t.add_resource(sns.Topic(
            TopicName="one-test",
            Subscriptions=))

        t.add_resource(sns.Topic(
            TopicName="another-test",
            Subscriptions=subscriptions))
```

A sample config for the above:

```
stacks:
  - name: buckets
    class_path: path.to.above.Buckets
    variables:
      Buckets:
        # resource name (title) that will be added to CloudFormation.
        FirstBucket:
          # name of the s3 bucket
          BucketName: my-first-bucket
        SecondBucket:
          BucketName: my-second-bucket
      SingleBucket:
        # resource name (title) that will be added to CloudFormation.
        MySingleBucket:
          BucketName: my-single-bucket
      Subscriptions:
        - Endpoint: one-lambda
          Protocol: lambda
        - Endpoint: another-lambda
          Protocol: lambda
      # The following could be ommited entirely
      SingleOptionalSubscription:
        Endpoint: a-third-lambda
        Protocol: lambda
```

CFNType

The `CFNType` can be used to signal that a variable should be submitted to CloudFormation as a Parameter instead of only available to the Blueprint when rendering. This is useful if you want to leverage AWS- Specific Parameter types (e.g. `List<AWS::EC2::Image::Id>`) or Systems Manager Parameter Store values (e.g. `AWS::SSM::Parameter::Value<String>`). See `stacker.blueprints.variables.types` for available subclasses of the `CFNType`.

Example

Below is an annotated example:

```
from stacker.blueprints.base import Blueprint
from stacker.blueprints.variables.types import (
    CFNString,
    EC2AvailabilityZoneNameList,
)


class SampleBlueprint(Blueprint):

    VARIABLES = {
        "String": {
            "type": str,
            "description": "Simple string variable",
        },
        "List": {
            "type": list,
```

(continues on next page)

(continued from previous page)

```
            "description": "Simple list variable",
        },
        "CloudFormationString": {
            "type": CFNString,
            "description": "A variable which will create a CloudFormation Parameter␣
↪of type String",
        },
        "CloudFormationSpecificType": {
            "type": EC2AvailabilityZoneNameList,
            "description": "A variable which will create a CloudFormation Parameter␣
↪of type List<AWS::EC2::AvailabilityZone::Name>"
        },
    }

    def create_template(self):
        t = self.template

        # `get_variables` returns a dictionary of <variable name>: <variable
        value>. For the subclasses of `CFNType`, the values are
        instances of `CFNParameter` which have a `ref` helper property
        which will return a troposphere `Ref` to the parameter name.
        variables = self.get_variables()

        t.add_output(Output("StringOutput", variables["String"]))

        # variables["List"] is a native list
        for index, value in enumerate(variables["List"]):
            t.add_output(Output("ListOutput:{}".format(index), value))


        # `CFNParameter` values (which wrap variables with a `type`
        that is a `CFNType` subclass) can be converted to troposphere
        `Ref` objects with the `ref` property
        t.add_output(Output("CloudFormationStringOutput",
                            variables["CloudFormationString"].ref))
        t.add_output(Output("CloudFormationSpecificTypeOutput",
                            variables["CloudFormationSpecificType"].ref))
```

1.8.3 Utilizing Stack name within your Blueprint

Sometimes your blueprint might want to utilize the already existing stack name within your blueprint. Stacker provides access to both the fully qualified stack name matching what's shown in the CloudFormation console, in addition to the stacks short name you have set in your YAML config.

Referencing Fully Qualified Stack name

The fully qualified name is a combination of the Stacker namespace + the short name (what you set as *name* in your YAML config file). If your stacker namespace is *StackerIsCool* and the stacks short name is *myAwesomeEC2Instance*, the fully qualified name would be:

`StackerIsCool-myAwesomeEC2Instance`

To use this in your blueprint, you can get the name from context. The `self.context.get_fqn(self.name)`

Referencing the Stack short name

The Stack short name is the name you specified for the stack within your YAML config. It does not include the namespace. If your stacker namespace is *StackerIsCool* and the stacks short name is *myAwesomeEC2Instance*, the short name would be:

`myAwesomeEC2Instance`

To use this in your blueprint, you can get the name from self.name: `self.name`

Example

Below is an annotated example creating a security group:

```
# we are importing Ref to allow for CFN References in the EC2 resource.  Tags
# will be used to set the Name tag
from troposphere import Ref, ec2, Tags
from stacker.blueprints.base import Blueprint
# CFNString is imported to allow for stand alone stack use
from stacker.blueprints.variables.types import CFNString

class SampleBlueprint(Blueprint):

  # VpcId set here to allow for blueprint to be reused
  VARIABLES = {
  "VpcId": {
      "type": CFNString,
      "description": "The VPC to create the Security group in",
      }
  }


  def create_template(self):
      template = self.template
      # Assigning the variables to a variable
      variables = self.get_variables()
      # now adding a SecurityGroup resource named `SecurityGroup` to the CFN template
      template.add_resource(
        ec2.SecurityGroup(
          "SecurityGroup",
          # Refering the VpcId set as the varible
          VpcId=variables['VpcId'].ref,
          # Setting the group description as the fully qualified name
          GroupDescription=self.context.get_fqn(self.name),
          # setting the Name tag to be the stack short name
          Tags=Tags(
            Name=self.name
            )
          )
        )
```

1.8.4 Testing Blueprints

When writing your own blueprints its useful to write tests for them in order to make sure they behave the way you expect they would, especially if there is any complex logic inside.

To this end, a sub-class of the *unittest.TestCase* class has been provided: *stacker.blueprints.testutil.BlueprintTestCase*. You use it like the regular TestCase class, but it comes with an addition assertion: *assertRenderedBlueprint*. This assertion takes a Blueprint object and renders it, then compares it to an expected output, usually in *tests/fixtures/blueprints*.

Examples of using the *BlueprintTestCase* class can be found in the stacker_blueprints repo. For example, see the tests used to test the Route53 DNSRecords Blueprint and the accompanying output results:

Yaml (stacker) format tests

In order to wrap the *BlueprintTestCase* tests in a format similar to stacker's stack format, the *YamlDirTestGenerator* class is provided. When subclassed in a directory, it will search for yaml files in that directory with certain structure and execute a test case for it. As an example:

```
---
namespace: test
stacks:
  - name: test_stack
    class_path: stacker_blueprints.s3.Buckets
    variables:
      var1: val1
```

When run from tests, this will create a template fixture file called test_stack.json containing the output from the *stacker_blueprints.s3.Buckets* template.

Examples of using the *YamlDirTestGenerator* class can be found in the stacker_blueprints repo. For example, see the tests used to test the s3.Buckets class and the accompanying fixture. These are generated from a subclass of YamlDirTestGenerator.

1.9 Templates

CloudFormation templates can be provided via python **Blueprints_** or JSON/YAML. JSON/YAML templates are specified for stacks via the `template_path` config option (see Stacks).

1.9.1 Jinja2 Templating

Templates with a `.j2` extension will be parsed using Jinja2. The stacker `context` and `mappings` objects and stack `variables` objects are available for use in the template:

```
Description: TestTemplate
Resources:
  Bucket:
    Type: AWS::S3::Bucket
    Properties:
      BucketName: {{ context.environment.foo }}-{{ variables.myparamname }}
```

1.10 stacker

1.10.1 stacker package

Subpackages

stacker.actions package

Submodules

stacker.actions.base module

class stacker.actions.base.**BaseAction**(*context*, *provider_builder=None*, *cancel=None*)

Bases: future.types.newobject.newobject

Actions perform the actual work of each Command.

Each action is tied to a stacker.commands.base.BaseCommand, and is responsible for building the *stacker.plan.Plan* that will be executed to perform that command.

Parameters **context** (*stacker.context.Context*) – The stacker context for the current run.

:param provider_builder (*stacker.providers.base.BaseProviderBuilder*,: optional): An object that will buil
interacted with in order to perform the necessary actions.

build_provider(*stack*)

Builds a stacker.providers.base.Provider suitable for operating on the given stacker.Stack.

ensure_cfn_bucket()

The CloudFormation bucket where templates will be stored.

execute(**args*, ***kwargs*)

post_run(**args*, ***kwargs*)

pre_run(**args*, ***kwargs*)

provider

Some actions need a generic provider using the default region (e.g. hooks).

run(**args*, ***kwargs*)

s3_stack_push(*blueprint*, *force=False*)

Pushes the rendered blueprint's template to S3.

Verifies that the template doesn't already exist in S3 before pushing.

Returns the URL to the template in S3.

stack_template_url(*blueprint*)

stacker.actions.base.**build_walker**(*concurrency*)

This will return a function suitable for passing to *stacker.plan.Plan* for walking the graph.

If concurrency is 1 (no parallelism) this will return a simple topological walker that doesn't use any multithreading.

If concurrency is 0, this will return a walker that will walk the graph as fast as the graph topology allows.

If concurrency is greater than 1, it will return a walker that will only execute a maximum of concurrency steps at any given time.

Returns returns a function to walk a stacker.dag.DAG.

Return type func

stacker.actions.base.**plan**(*description*, *stack_action*, *context*, *tail=None*, *reverse=False*)
A simple helper that builds a graph based plan from a set of stacks.

Parameters

- **description** (*str*) – a description of the plan.
- **action** (*func*) – a function to call for each stack.
- **context** (*stacker.context.Context*) – a *stacker.context.Context* to build the plan from.
- **tail** (*func*) – an optional function to call to tail the stack progress.
- **reverse** (*bool*) – if True, execute the graph in reverse (useful for destroy actions).

Returns The resulting plan object

Return type plan.Plan

stacker.actions.base.**stack_template_key_name**(*blueprint*)
Given a blueprint, produce an appropriate key name.

Parameters **blueprint** (*stacker.blueprints.base.Blueprint*) – The blueprint object to create the key from.

Returns Key name resulting from blueprint.

Return type string

stacker.actions.base.**stack_template_url**(*bucket_name*, *blueprint*, *endpoint*)
Produces an s3 url for a given blueprint.

Parameters

- **bucket_name** (*string*) – The name of the S3 bucket where the resulting templates are stored.
- **blueprint** (*stacker.blueprints.base.Blueprint*) – The blueprint object to create the URL to.
- **endpoint** (*string*) – The s3 endpoint used for the bucket.

Returns S3 URL.

Return type string

stacker.actions.build module

class stacker.actions.build.**Action**(*context*, *provider_builder=None*, *cancel=None*)
Bases: *stacker.actions.base.BaseAction*

Responsible for building & coordinating CloudFormation stacks.

Generates the build plan based on stack dependencies (these dependencies are determined automatically based on output lookups from other stacks).

The plan can then either be printed out as an outline or executed. If executed, each stack will get launched in order which entails:

- Pushing the generated CloudFormation template to S3 if it has changed
- **Submitting either a build or update of the given stack to the** stacker.provider.base.Provider.

build_parameters (*stack*, *provider_stack=None*)
Builds the CloudFormation Parameters for our stack.

Parameters

- **stack** (*stacker.stack.Stack*) – A stacker stack
- **provider_stack** (*dict*) – An optional Stacker provider object

Returns The parameters for the given stack

Return type dict

post_run (*outline=False*, *dump=False*, **args*, ***kwargs*)
Any steps that need to be taken after running the action.

pre_run (*outline=False*, *dump=False*, **args*, ***kwargs*)
Any steps that need to be taken prior to running the action.

run (*concurrency=0*, *outline=False*, *tail=False*, *dump=False*, **args*, ***kwargs*)
Kicks off the build/update of the stacks in the stack_definitions.

This is the main entry point for the Builder.

class stacker.actions.build.**UsePreviousParameterValue**
Bases: object

A simple class used to indicate a Parameter should use it's existng value.

stacker.actions.build.**build_stack_tags** (*stack*)
Builds a common set of tags to attach to a stack

stacker.actions.build.**handle_hooks** (*stage*, *hooks*, *provider*, *context*, *dump*, *outline*)
Handle pre/post hooks.

Parameters

- **stage** (*str*) – The name of the hook stage - pre_build/post_build.
- **hooks** (*list*) – A list of dictionaries containing the hooks to execute.
- **provider** (stacker.provider.base.BaseProvider) – The provider the current stack is using.
- **context** (*stacker.context.Context*) – The current stacker context.
- **dump** (*bool*) – Whether running with dump set or not.
- **outline** (*bool*) – Whether running with outline set or not.

stacker.actions.build.**should_ensure_cfn_bucket** (*outline*, *dump*)
Test whether access to the cloudformation template bucket is required

Parameters

- **outline** (*bool*) – The outline action.
- **dump** (*bool*) – The dump action.

Returns If access to CF bucket is needed, return True.

Return type bool

stacker.actions.build.**should_submit** (*stack*)
Tests whether a stack should be submitted to CF for update/create

Parameters **stack** (*stacker.stack.Stack*) – The stack object to check.

Returns If the stack should be submitted, return True.

Return type bool

stacker.actions.build.**should_update**(*stack*)
Tests whether a stack should be submitted for updates to CF.

Parameters **stack** (*stacker.stack.Stack*) – The stack object to check.

Returns If the stack should be updated, return True.

Return type bool

stacker.actions.destroy module

class stacker.actions.destroy.**Action**(*context*, *provider_builder=None*, *cancel=None*)
Bases: *stacker.actions.base.BaseAction*

Responsible for destroying CloudFormation stacks.

Generates a destruction plan based on stack dependencies. Stack dependencies are reversed from the build action. For example, if a Stack B requires Stack A during build, during destroy Stack A requires Stack B be destroyed first.

The plan defaults to printing an outline of what will be destroyed. If forced to execute, each stack will get destroyed in order.

post_run(*outline=False*, **args*, ***kwargs*)
Any steps that need to be taken after running the action.

pre_run(*outline=False*, **args*, ***kwargs*)
Any steps that need to be taken prior to running the action.

run(*force*, *concurrency=0*, *tail=False*, **args*, ***kwargs*)

stacker.actions.diff module

class stacker.actions.diff.**Action**(*context*, *provider_builder=None*, *cancel=None*)
Bases: *stacker.actions.build.Action*

Responsible for diff'ing CF stacks in AWS and on disk

Generates the build plan based on stack dependencies (these dependencies are determined automatically based on references to output values from other stacks).

The plan is then used to create a changeset for a stack using a generated template based on the current config.

post_run(**args*, ***kwargs*)
Any steps that need to be taken after running the action.

pre_run(**args*, ***kwargs*)
Any steps that need to be taken prior to running the action.

run(*concurrency=0*, **args*, ***kwargs*)
Kicks off the build/update of the stacks in the stack_definitions.

This is the main entry point for the Builder.

class stacker.actions.diff.**DictValue**(*key*, *old_value*, *new_value*)
Bases: future.types.newobject.newobject

ADDED = 'ADDED'

MODIFIED = 'MODIFIED'

REMOVED = 'REMOVED'

UNMODIFIED = 'UNMODIFIED'

changes()
Returns a list of changes to represent the diff between old and new value.

Returns

[string] representation of the change (if any) between old and new value

Return type list

formatter = '%s%s = %s'

status()

stacker.actions.diff.**diff_dictionaries**(*old_dict*, *new_dict*)
Diffs two single dimension dictionaries

Returns the number of changes and an unordered list expressing the common entries and changes.

Parameters

- **old_dict** (*dict*) – old dictionary
- **new_dict** (*dict*) – new dictionary

Returns: list() int: number of changed records list: [DictValue]

stacker.actions.diff.**diff_parameters**(*old_params*, *new_params*)
Compares the old vs. new parameters and returns a "diff"

If there are no changes, we return an empty list.

Parameters

- **old_params** (*dict*) – old paramters
- **new_params** (*dict*) – new parameters

Returns A list of differences

Return type list

stacker.actions.diff.**format_params_diff**(*parameter_diff*)
Handles the formatting of differences in parameters.

Parameters **parameter_diff** (*list*) – A list of DictValues detailing the differences between two dicts returned by *stacker.actions.diff.diff_dictionaries()*

Returns A formatted string that represents a parameter diff

Return type string

stacker.actions.info module

class stacker.actions.info.**Action**(*context*, *provider_builder=None*, *cancel=None*)
Bases: *stacker.actions.base.BaseAction*

Get information on CloudFormation stacks.

Displays the outputs for the set of CloudFormation stacks.

run(**args*, ***kwargs*)

Module contents

stacker.blueprints package

Subpackages

stacker.blueprints.variables package

Submodules

stacker.blueprints.variables.types module

class stacker.blueprints.variables.types.**CFNType**(*parameter_type*)
Bases: future.types.newobject.newobject

class stacker.blueprints.variables.types.**TroposphereType**(*defined_type*, *many=False*, *optional=False*, *validate=True*)
Bases: future.types.newobject.newobject

create(*value*)
Create the troposphere type from the value.

Parameters **value** (*Union[dict, list]*) – A dictionary or list of dictionaries (see class documentation for details) to use as parameters to create the Troposphere type instance. Each dictionary will be passed to the *from_dict* method of the type.

Returns

Returns the value converted to the troposphere type

Return type Union[list, type]

resource_name

Module contents

Submodules

stacker.blueprints.base module

class stacker.blueprints.base.**Blueprint**(*name*, *context*, *mappings=None*, *description=None*)
Bases: future.types.newobject.newobject

Base implementation for rendering a troposphere template.

Parameters

- **name** (*str*) – A name for the blueprint.
- **context** (*stacker.context.Context*) – the context the blueprint is being executed under.

- **mappings** (*dict, optional*) – Cloudformation Mappings to be used in the template.

add_output(*name*, *value*)
Simple helper for adding outputs.

Parameters

- **name** (*str*) – The name of the output to create.
- **value** (*str*) – The value to put in the output.

create_template()

defined_variables()
Return a dictionary of variables defined by the blueprint.

By default, this will just return the values from *VARIABLES*, but this makes it easy for subclasses to add variables.

Returns variables defined by the blueprint

Return type dict

get_cfn_parameters()
Return a dictionary of variables with *type* `CFNType`.

Returns

variables that need to be submitted as CloudFormation Parameters.

Return type dict

get_output_definitions()
Gets the output definitions.

Returns

output definitions. Keys are output names, the values are dicts containing key/values for various output properties.

Return type dict

get_parameter_definitions()
Get the parameter definitions to submit to CloudFormation.

Any variable definition whose *type* is an instance of *CFNType* will be returned as a CloudFormation Parameter.

Returns

parameter definitions. Keys are parameter names, the values are dicts containing key/values for various parameter properties.

Return type dict

get_parameter_values()
Return a dictionary of variables with *type* `CFNType`.

Returns

variables that need to be submitted as CloudFormation Parameters. Will be a dictionary of <parameter name>: <parameter value>.

Return type dict

get_required_parameter_definitions()
Returns all template parameters that do not have a default value.

Returns

dict of required CloudFormation Parameters for the blueprint. Will be a dictionary of <parameter name>: <parameter attributes>.

Return type dict

get_variables()
Return a dictionary of variables available to the template.

These variables will have been defined within *VARIABLES* or *self.defined_variables*. Any variable value that contains a lookup will have been resolved.

Returns variables available to the template

Return type dict

Raises:

import_mappings()

read_user_data(*user_data_path*)
Reads and parses a user_data file.

Parameters **user_data_path** (*str*) – path to the userdata file

Returns the parsed user data file

Return type str

render_template()
Render the Blueprint to a CloudFormation template

rendered

requires_change_set
Returns true if the underlying template has transforms.

reset_template()

resolve_variables(*provided_variables*)
Resolve the values of the blueprint variables.

This will resolve the values of the *VARIABLES* with values from the env file, the config, and any lookups resolved.

Parameters **provided_variables** (list of *stacker.variables.Variable*) – list of provided variables

set_template_description(*description*)
Adds a description to the Template

Parameters **description** (*str*) – A description to be added to the resulting template.

setup_parameters()
Add any CloudFormation parameters to the template

to_json(*variables=None*)
Render the blueprint and return the template in json form.

Parameters **variables** (*dict*) – Optional dictionary providing/overriding variable values.

Returns the rendered CFN JSON template

Return type str

version

class stacker.blueprints.base.**CFNParameter**(*name*, *value*)
Bases: future.types.newobject.newobject

ref

to_parameter_value()
Return the value to be submitted to CloudFormation

stacker.blueprints.base.**build_parameter**(*name*, *properties*)
Builds a troposphere Parameter with the given properties.

Parameters

- **name** (*string*) – The name of the parameter.
- **properties** (*dict*) – Contains the properties that will be applied to the parameter. See: http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/parameters-section-structure.html

Returns The created parameter object.

Return type troposphere.Parameter

stacker.blueprints.base.**parse_user_data**(*variables*, *raw_user_data*, *blueprint_name*)
Parse the given user data and renders it as a template

It supports referencing template variables to create userdata that's supplemented with information from the stack, as commonly required when creating EC2 userdata files.

For example: Given a raw_user_data string: 'open file ${file}' And a variables dictionary with: {'file': 'test.txt'} parse_user_data would output: open file test.txt

Parameters

- **variables** (*dict*) – variables available to the template
- **raw_user_data** (*str*) – the user_data to be parsed
- **blueprint_name** (*str*) – the name of the blueprint

Returns

The parsed user data, with all the variables values and refs replaced with their resolved values.

Return type str

Raises

- InvalidUserdataPlaceholder – Raised when a placeholder name in raw_user_data is not valid. E.g ${100} would raise this.
- MissingVariable – Raised when a variable is in the raw_user_data that is not given in the blueprint

stacker.blueprints.base.**resolve_variable**(*var_name*, *var_def*, *provided_variable*, *blueprint_name*)
Resolve a provided variable value against the variable definition.

Parameters

- **var_name** (*str*) – The name of the defined variable on a blueprint.
- **var_def** (*dict*) – A dictionary representing the defined variables attributes.

- **provided_variable** (*stacker.variables.Variable*) – The variable value provided to the blueprint.
- **blueprint_name** (*str*) – The name of the blueprint that the variable is being applied to.

Returns The resolved variable value, could be any python object.

Return type object

Raises

- `MissingVariable` – Raised when a variable with no default is not provided a value.
- `UnresolvedVariable` – Raised when the provided variable is not already resolved.
- `ValueError` – Raised when the value is not the right type and cannot be cast as the correct type. Raised by *stacker.blueprints.base.validate_variable_type()*
- `ValidatorError` – Raised when a validator raises an exception. Wraps the original exception.

`stacker.blueprints.base.`**validate_allowed_values**(*allowed_values*, *value*)

Support a variable defining which values it allows.

Parameters

- **allowed_values** (*Optional[list]*) – A list of allowed values from the variable definition
- **value** (*obj*) – The object representing the value provided for the variable

Returns Boolean for whether or not the value is valid.

Return type bool

`stacker.blueprints.base.`**validate_variable_type**(*var_name*, *var_type*, *value*)

Ensures the value is the correct variable type.

Parameters

- **var_name** (*str*) – The name of the defined variable on a blueprint.
- **var_type** (*type*) – The type that the value should be.
- **value** (*obj*) – The object representing the value provided for the variable

Returns

Returns the appropriate value object. If the original value was of CFNType, the returned value will be wrapped in CFNParameter.

Return type object

Raises `ValueError` – If the *value* isn't of *var_type* and can't be cast as that type, this is raised.

stacker.blueprints.testutil module

class `stacker.blueprints.testutil.`**BlueprintTestCase**(*methodName='runTest'*)

Bases: `unittest.case.TestCase`

OUTPUT_PATH = 'tests/fixtures/blueprints'

assertRenderedBlueprint(*blueprint*)

class stacker.blueprints.testutil.**YamlDirTestGenerator**
Bases: object

Generate blueprint tests from yaml config files.

This class creates blueprint tests from yaml files with a syntax similar to stackers' configuration syntax. For example,

— namespace: test stacks:

- name: test_sample class_path: stacker_blueprints.test.Sample variables:

var1: value1

will create a test for the specified blueprint, passing that variable as part of the test.

The test will generate a .json file for this blueprint, and compare it with the stored result.

By default, the generator looks for files named 'test_*.yaml' in its same directory. In order to use it, subclass it in a directory containing such tests, and name the class with a pattern that will include it in nosetests' tests (for example, TestGenerator).

The subclass may override some properties:

@property base_class: by default, the generated tests are subclasses of stacker.blueprints.testutil.BlueprintTestCase. In order to change this, set this property to the desired base class.

@property yaml_dirs: by default, the directory where the generator is subclassed is searched for test files. Override this array for specifying more directories. These must be relative to the directory in which the subclass lives in. Globs may be used.

Default: ['.']. Example override: ['.', 'tests/*/']

@property yaml_filename: by default, the generator looks for files named 'test_*.yaml'. Use this to change this pattern. Globs may be used.

There's an example of this use in the tests/ subdir of stacker_blueprints.

base_class

test_generator()

yaml_dirs

yaml_filename

stacker.blueprints.testutil.**diff**(*a*, *b*)
A human readable differ.

Module contents

stacker.commands package

Subpackages

stacker.commands.stacker package

Submodules

stacker.commands.stacker.base module

class stacker.commands.stacker.base.**BaseCommand**(*setup_logging=None*, **args*, ***kwargs*)

Bases: future.types.newobject.newobject

Base class for all stacker subcommands.

The way argparse handles common arguments that should be passed to the subparser is confusing. You can add arguments to the parent parser that will get passed to the subparser, but these then need to be provided on the command line before specifying the subparser. Furthermore, when viewing the help for a subcommand, you can't view these parameters.

By including shared parameters for stacker commands within this subclass, we don't have to redundantly add the parameters we want on all subclasses within each subparser and these shared parameters are treated as normal arguments to the subcommand.

add_arguments(*parser*)

add_subcommands(*parser*)

configure(*options*, ***kwargs*)

description = None

get_context_kwargs(*options*, ***kwargs*)

Return a dictionary of kwargs that will be used with the Context.

This allows commands to pass in any specific arguments they define to the context.

Parameters **options** (argparse.Namespace) – arguments that have been passed via the command line

Returns

Dictionary that will be passed to Context initializer as kwargs.

Return type dict

name = None

parse_args(**vargs*)

run(*options*, ***kwargs*)

subcommands = ()

subcommands_help = None

class stacker.commands.stacker.base.**KeyValueAction**(*option_strings*, *dest*, *default=None*, *nargs=None*, ***kwargs*)

Bases: argparse.Action

stacker.commands.stacker.base.**cancel**()

Returns a threading.Event() that will get set when SIGTERM, or SIGINT are triggered. This can be used to cancel execution of threads.

stacker.commands.stacker.base.**environment_file**(*input_file*)

Reads a stacker environment file and returns the resulting data.

stacker.commands.stacker.base.**key_value_arg**(*string*)

stacker.commands.stacker.build module

Launches or updates CloudFormation stacks based on the given config.

Stacker is smart enough to figure out if anything (the template or parameters) have changed for a given stack. If nothing has changed, stacker will correctly skip executing anything against the stack.

class stacker.commands.stacker.build.**Build**(*setup_logging=None*, **args*, ***kwargs*)
Bases: *stacker.commands.stacker.base.BaseCommand*

add_arguments(*parser*)

description = 'Launches or updates CloudFormation stacks based on the given config.\n

get_context_kwargs(*options*, ***kwargs*)
Return a dictionary of kwargs that will be used with the Context.

This allows commands to pass in any specific arguments they define to the context.

Parameters **options** (argparse.Namespace) – arguments that have been passed via the command line

Returns

Dictionary that will be passed to Context initializer as kwargs.

Return type dict

name = 'build'

run(*options*, ***kwargs*)

stacker.commands.stacker.destroy module

Destroys CloudFormation stacks based on the given config.

Stacker will determine the order in which stacks should be destroyed based on any manual requirements they specify or output values they rely on from other stacks.

class stacker.commands.stacker.destroy.**Destroy**(*setup_logging=None*, **args*, ***kwargs*)
Bases: *stacker.commands.stacker.base.BaseCommand*

add_arguments(*parser*)

description = 'Destroys CloudFormation stacks based on the given config.\n\nStacker wi

get_context_kwargs(*options*, ***kwargs*)
Return a dictionary of kwargs that will be used with the Context.

This allows commands to pass in any specific arguments they define to the context.

Parameters **options** (argparse.Namespace) – arguments that have been passed via the command line

Returns

Dictionary that will be passed to Context initializer as kwargs.

Return type dict

name = 'destroy'

run(*options*, ***kwargs*)

stacker.commands.stacker.diff module

Diffs the config against the currently running CloudFormation stacks

Sometimes small changes can have big impacts. Run "stacker diff" before "stacker build" to detect bad things(tm) from happening in advance!

class stacker.commands.stacker.diff.**Diff**(*setup_logging=None*, **args*, ***kwargs*)
Bases: *stacker.commands.stacker.base.BaseCommand*

add_arguments(*parser*)

description = ' Diffs the config against the currently running CloudFormation stacks\n

get_context_kwargs(*options*, ***kwargs*)
Return a dictionary of kwargs that will be used with the Context.

This allows commands to pass in any specific arguments they define to the context.

Parameters **options** (argparse.Namespace) – arguments that have been passed via the command line

Returns

Dictionary that will be passed to Context initializer as kwargs.

Return type dict

name = 'diff'

run(*options*, ***kwargs*)

stacker.commands.stacker.info module

Gets information on the CloudFormation stacks based on the given config.

class stacker.commands.stacker.info.**Info**(*setup_logging=None*, **args*, ***kwargs*)
Bases: *stacker.commands.stacker.base.BaseCommand*

add_arguments(*parser*)

description = 'Gets information on the CloudFormation stacks based on the given config

get_context_kwargs(*options*, ***kwargs*)
Return a dictionary of kwargs that will be used with the Context.

This allows commands to pass in any specific arguments they define to the context.

Parameters **options** (argparse.Namespace) – arguments that have been passed via the command line

Returns

Dictionary that will be passed to Context initializer as kwargs.

Return type dict

name = 'info'

run(*options*, ***kwargs*)

Module contents

class stacker.commands.stacker.**Stacker**(*setup_logging=None*, **args*, ***kwargs*)

Bases: *stacker.commands.stacker.base.BaseCommand*

add_arguments(*parser*)

configure(*options*, ***kwargs*)

name = 'stacker'

subcommands = (<class 'stacker.commands.stacker.build.Build'>, <class 'stacker.commands

Module contents

stacker.config package

Subpackages

stacker.config.translators package

Submodules

stacker.config.translators.kms module

stacker.config.translators.kms.**kms_simple_constructor**(*loader*, *node*)

Module contents

Module contents

class stacker.config.**AnyType**(*required=False*, *default=Undefined*, *serialized_name=None*, *choices=None*, *validators=None*, *deserialize_from=None*, *export_level=None*, *serialize_when_none=None*, *messages=None*, *metadata=None*)

Bases: schematics.types.base.BaseType

MESSAGES = {u'choices': u'Value must be one of {0}.', u'required': u'This field is r

class stacker.config.**Config**(*raw_data=None*, *trusted_data=None*, *deserialize_mapping=None*, *init=True*, *partial=True*, *strict=True*, *validate=False*, *app_data=None*, *lazy=False*, ***kwargs*)

Bases: schematics.deprecated.Model

This is the Python representation of a stacker config file.

This is used internally by stacker to parse and validate a yaml formatted stacker configuration file, but can also be used in scripts to generate a stacker config file before handing it off to stacker to build/destroy.

Example:

```
from stacker.config import dump, Config, Stack

vpc = Stack({
```

(continues on next page)

(continued from previous page)

```
    "name": "vpc",
    "class_path": "blueprints.VPC"})

config = Config()
config.namespace = "prod"
config.stacks = [vpc]

print dump(config)
```

log_formats = <DictType(StringType) instance on Config as 'log_formats'>

lookups = <DictType(StringType) instance on Config as 'lookups'>

mappings = <DictType(DictType) instance on Config as 'mappings'>

namespace = <StringType() instance on Config as 'namespace'>

namespace_delimiter = <StringType() instance on Config as 'namespace_delimiter'>

package_sources = <ModelType(PackageSources) instance on Config as 'package_sources'>

post_build = <ListType(ModelType) instance on Config as 'post_build'>

post_destroy = <ListType(ModelType) instance on Config as 'post_destroy'>

pre_build = <ListType(ModelType) instance on Config as 'pre_build'>

pre_destroy = <ListType(ModelType) instance on Config as 'pre_destroy'>

service_role = <StringType() instance on Config as 'service_role'>

stacker_bucket = <StringType() instance on Config as 'stacker_bucket'>

stacker_bucket_region = <StringType() instance on Config as 'stacker_bucket_region'>

stacker_cache_dir = <StringType() instance on Config as 'stacker_cache_dir'>

stacks = <ListType(ModelType) instance on Config as 'stacks'>

sys_path = <StringType() instance on Config as 'sys_path'>

tags = <DictType(StringType) instance on Config as 'tags'>

targets = <ListType(ModelType) instance on Config as 'targets'>

template_indent = <StringType() instance on Config as 'template_indent'>

validate(**args*, ***kwargs*)

validate_stacks(*data*, *value*)

class stacker.config.**GitPackageSource**(*raw_data=None*, *trusted_data=None*, *deserialize_mapping=None*, *init=True*, *partial=True*, *strict=True*, *validate=False*, *app_data=None*, *lazy=False*, ***kwargs*)

Bases: schematics.deprecated.Model

branch = <StringType() instance on GitPackageSource as 'branch'>

commit = <StringType() instance on GitPackageSource as 'commit'>

configs = <ListType(StringType) instance on GitPackageSource as 'configs'>

paths = <ListType(StringType) instance on GitPackageSource as 'paths'>

tag = <StringType() instance on GitPackageSource as 'tag'>

uri = <StringType() instance on GitPackageSource as 'uri'>

class stacker.config.**Hook**(*raw_data=None*, *trusted_data=None*, *deserialize_mapping=None*, *init=True*, *partial=True*, *strict=True*, *validate=False*, *app_data=None*, *lazy=False*, ***kwargs*)

Bases: schematics.deprecated.Model

args = <DictType(AnyType) instance on Hook as 'args'>

data_key = <StringType() instance on Hook as 'data_key'>

enabled = <BooleanType() instance on Hook as 'enabled'>

path = <StringType() instance on Hook as 'path'>

required = <BooleanType() instance on Hook as 'required'>

class stacker.config.**LocalPackageSource**(*raw_data=None*, *trusted_data=None*, *deserialize_mapping=None*, *init=True*, *partial=True*, *strict=True*, *validate=False*, *app_data=None*, *lazy=False*, ***kwargs*)

Bases: schematics.deprecated.Model

configs = <ListType(StringType) instance on LocalPackageSource as 'configs'>

paths = <ListType(StringType) instance on LocalPackageSource as 'paths'>

source = <StringType() instance on LocalPackageSource as 'source'>

class stacker.config.**PackageSources**(*raw_data=None*, *trusted_data=None*, *deserialize_mapping=None*, *init=True*, *partial=True*, *strict=True*, *validate=False*, *app_data=None*, *lazy=False*, ***kwargs*)

Bases: schematics.deprecated.Model

git = <ListType(ModelType) instance on PackageSources as 'git'>

local = <ListType(ModelType) instance on PackageSources as 'local'>

s3 = <ListType(ModelType) instance on PackageSources as 's3'>

class stacker.config.**S3PackageSource**(*raw_data=None*, *trusted_data=None*, *deserialize_mapping=None*, *init=True*, *partial=True*, *strict=True*, *validate=False*, *app_data=None*, *lazy=False*, ***kwargs*)

Bases: schematics.deprecated.Model

bucket = <StringType() instance on S3PackageSource as 'bucket'>

configs = <ListType(StringType) instance on S3PackageSource as 'configs'>

key = <StringType() instance on S3PackageSource as 'key'>

paths = <ListType(StringType) instance on S3PackageSource as 'paths'>

requester_pays = <BooleanType() instance on S3PackageSource as 'requester_pays'>

use_latest = <BooleanType() instance on S3PackageSource as 'use_latest'>

class stacker.config.**Stack**(*raw_data=None*, *trusted_data=None*, *deserialize_mapping=None*, *init=True*, *partial=True*, *strict=True*, *validate=False*, *app_data=None*, *lazy=False*, ***kwargs*)

Bases: schematics.deprecated.Model

class_path = <StringType() instance on Stack as 'class_path'>

description = <StringType() instance on Stack as 'description'>

enabled = <BooleanType() instance on Stack as 'enabled'>

in_progress_behavior = <StringType() instance on Stack as 'in_progress_behavior'>

locked = <BooleanType() instance on Stack as 'locked'>

name = <StringType() instance on Stack as 'name'>

notification_arns = <ListType(StringType) instance on Stack as 'notification_arns'>

parameters = <DictType(AnyType) instance on Stack as 'parameters'>

profile = <StringType() instance on Stack as 'profile'>

protected = <BooleanType() instance on Stack as 'protected'>

region = <StringType() instance on Stack as 'region'>

required_by = <ListType(StringType) instance on Stack as 'required_by'>

requires = <ListType(StringType) instance on Stack as 'requires'>

stack_name = <StringType() instance on Stack as 'stack_name'>

stack_policy_path = <StringType() instance on Stack as 'stack_policy_path'>

tags = <DictType(StringType) instance on Stack as 'tags'>

template_path = <StringType() instance on Stack as 'template_path'>

validate_class_path(*data*, *value*)

validate_parameters(*data*, *value*)

validate_stack_source(*data*)

validate_template_path(*data*, *value*)

variables = <DictType(AnyType) instance on Stack as 'variables'>

class stacker.config.**Target**(*raw_data=None*, *trusted_data=None*, *deserialize_mapping=None*, *init=True*, *partial=True*, *strict=True*, *validate=False*, *app_data=None*, *lazy=False*, ***kwargs*)

Bases: schematics.deprecated.Model

name = <StringType() instance on Target as 'name'>

required_by = <ListType(StringType) instance on Target as 'required_by'>

requires = <ListType(StringType) instance on Target as 'requires'>

stacker.config.**dump**(*config*)

Dumps a stacker Config object as yaml.

Parameters

- **config** (*Config*) – the stacker Config object.
- **stream** (*stream*) – an optional stream object to write to.

Returns the yaml formatted stacker Config.

Return type str

stacker.config.**load**(*config*)

Loads a stacker configuration by modifying sys paths, loading lookups, etc.

Parameters **config** (*Config*) – the stacker config to load.

Returns the stacker config provided above.

Return type *Config*

stacker.config.**not_empty_list**(*value*)

stacker.config.**parse**(*raw_config*)
Parse a raw yaml formatted stacker config.

Parameters **raw_config** (*str*) – the raw stacker configuration string in yaml format.

Returns the parsed stacker config.

Return type *Config*

stacker.config.**process_remote_sources**(*raw_config*, *environment=None*)
Stage remote package sources and merge in remote configs.

Parameters

- **raw_config** (*str*) – the raw stacker configuration string.
- **environment** (*dict, optional*) – any environment values that should be passed to the config

Returns the raw stacker configuration string

Return type str

stacker.config.**render**(*raw_config*, *environment=None*)
Renders a config, using it as a template with the environment.

Parameters

- **raw_config** (*str*) – the raw stacker configuration string.
- **environment** (DictWithSourceType, *optional*) – any environment values that should be passed to the config

Returns

the stacker configuration populated with any values passed from the environment

Return type str

stacker.config.**render_parse_load**(*raw_config*, *environment=None*, *validate=True*)
Encapsulates the render -> parse -> validate -> load process.

Parameters

- **raw_config** (*str*) – the raw stacker configuration string.
- **environment** (*dict, optional*) – any environment values that should be passed to the config
- **validate** (*bool*) – if provided, the config is validated before being loaded.

Returns the parsed stacker config.

Return type *Config*

stacker.config.**substitute_references**(*root*, *environment*, *exp*, *full_exp*)

stacker.hooks package

Submodules

stacker.hooks.aws_lambda module

stacker.hooks.aws_lambda.**select_bucket_region**(*custom_bucket*, *hook_region*, *stacker_bucket_region*, *provider_region*)

Returns the appropriate region to use when uploading functions.

Select the appropriate region for the bucket where lambdas are uploaded in.

Parameters

- **custom_bucket** (*str, None*) – The custom bucket name provided by the *bucket* kwarg of the aws_lambda hook, if provided.
- **hook_region** (*str*) – The contents of the *bucket_region* argument to the hook.
- **stacker_bucket_region** (*str*) – The contents of the *stacker_bucket_region* global setting.
- **provider_region** (*str*) – The region being used by the provider.

Returns The appropriate region string.

Return type str

stacker.hooks.aws_lambda.**upload_lambda_functions**(*context*, *provider*, ***kwargs*)

Builds Lambda payloads from user configuration and uploads them to S3.

Constructs ZIP archives containing files matching specified patterns for each function, uploads the result to Amazon S3, then stores objects (of type `troposphere.awslambda.Code`) in the context's hook data, ready to be referenced in blueprints.

Configuration consists of some global options, and a dictionary of function specifications. In the specifications, each key indicating the name of the function (used for generating names for artifacts), and the value determines what files to include in the ZIP (see more details below).

Payloads are uploaded to either a custom bucket or stackers default bucket, with the key containing it's checksum, to allow repeated uploads to be skipped in subsequent runs.

The configuration settings are documented as keyword arguments below.

Keyword Arguments

- **bucket** (*str, optional*) – Custom bucket to upload functions to. Omitting it will cause the default stacker bucket to be used.
- **bucket_region** (*str, optional*) – The region in which the bucket should exist. If not given, the region will be either be that of the global *stacker_bucket_region* setting, or else the region in use by the provider.
- **prefix** (*str, optional*) – S3 key prefix to prepend to the uploaded zip name.
- **follow_symlinks** (*bool, optional*) – Will determine if symlinks should be followed and included with the zip artifact. Default: False
- **payload_acl** (*str, optional*) – The canned S3 object ACL to be applied to the uploaded payload. Default: private
- **functions** (*dict*) – Configurations of desired payloads to build. Keys correspond to function names, used to derive key names for the payload. Each value should itself be a dictionary, with the following data:
  - path (str):

Base directory or path of a ZIP file of the Lambda function payload content.

If it not an absolute path, it will be considered relative to the directory containing the stacker configuration file in use.

When a directory, files contained will be added to the payload ZIP, according to the include and exclude patterns. If not patterns are provided, all files in the directory (respecting default exclusions) will be used.

Files are stored in the archive with path names relative to this directory. So, for example, all the files contained directly under this directory will be added to the root of the ZIP file.

When a ZIP file, it will be uploaded directly to S3. The hash of whole ZIP file will be used as the version key by default, which may cause spurious rebuilds when building the ZIP in different environments. To avoid that, explicitly provide a *version* option.

- include(str or list[str], optional):

  Pattern or list of patterns of files to include in the payload. If provided, only files that match these patterns will be included in the payload.

  Omitting it is equivalent to accepting all files that are not otherwise excluded.

- **exclude(str or list[str], optional):** Pattern or list of patterns of files to exclude from the payload. If provided, any files that match will be ignored, regardless of whether they match an inclusion pattern.

  Commonly ignored files are already excluded by default, such as `.git`, `.svn`, `__pycache__`, `*.pyc`, `.gitignore`, etc.

- **version(str, optional):** Value to use as the version for the current function, which will be used to determine if a payload already exists in S3. The value can be any string, such as a version number or a git commit.

  Note that when setting this value, to re-build/re-upload a payload you must change the version manually.

Examples

```
pre_build:
  - path: stacker.hooks.aws_lambda.upload_lambda_functions
    required: true
    enabled: true
    data_key: lambda
    args:
      bucket: custom-bucket
      follow_symlinks: true
      prefix: cloudformation-custom-resources/
      payload_acl: authenticated-read
      functions:
        MyFunction:
          path: ./lambda_functions
          include:
            - '*.py'
            - '*.txt'
          exclude:
            - '*.pyc'
            - test/
```

```
from troposphere.awslambda import Function
from stacker.blueprints.base import Blueprint

class LambdaBlueprint(Blueprint):
    def create_template(self):
        code = self.context.hook_data['lambda']['MyFunction']

        self.template.add_resource(
            Function(
                'MyFunction',
                Code=code,
                Handler='my_function.handler',
                Role='...',
                Runtime='python2.7'
            )
        )
```

stacker.hooks.ecs module

stacker.hooks.ecs.**create_clusters**(*provider*, *context*, ***kwargs*)

Creates ECS clusters.

Expects a "clusters" argument, which should contain a list of cluster names to create.

Parameters

- **provider** (*stacker.providers.base.BaseProvider*) – provider instance
- **context** (*stacker.context.Context*) – context instance

Returns: boolean for whether or not the hook succeeded.

stacker.hooks.iam module

stacker.hooks.iam.**create_ecs_service_role**(*provider*, *context*, ***kwargs*)

Used to create the ecsServieRole, which has to be named exactly that currently, so cannot be created via CloudFormation. See:

http://docs.aws.amazon.com/AmazonECS/latest/developerguide/IAM_policies.html#service_IAM_role

Parameters

- **provider** (*stacker.providers.base.BaseProvider*) – provider instance
- **context** (*stacker.context.Context*) – context instance

Returns: boolean for whether or not the hook succeeded.

stacker.hooks.iam.**ensure_server_cert_exists**(*provider*, *context*, ***kwargs*)

stacker.hooks.iam.**get_cert_contents**(*kwargs*)

Builds parameters with server cert file contents.

Parameters **kwargs** (*dict*) – The keyword args passed to ensure_server_cert_exists, optionally containing the paths to the cert, key and chain files.

Returns

A dictionary containing the appropriate parameters to supply to upload_server_certificate. An empty dictionary if there is a problem.

Return type dict

stacker.hooks.keypair module

stacker.hooks.keypair.**create_key_pair**(*ec2*, *keypair_name*)

stacker.hooks.keypair.**create_key_pair_from_public_key_file**(*ec2*, *keypair_name*, *public_key_path*)

stacker.hooks.keypair.**create_key_pair_in_ssm**(*ec2*, *ssm*, *keypair_name*, *parameter_name*, *kms_key_id=None*)

stacker.hooks.keypair.**create_key_pair_local**(*ec2*, *keypair_name*, *dest_dir*)

stacker.hooks.keypair.**ensure_keypair_exists**(*provider*, *context*, ***kwargs*)

Ensure a specific keypair exists within AWS.

If the key doesn't exist, upload it.

Parameters

- **provider** (*stacker.providers.base.BaseProvider*) – provider instance
- **context** (*stacker.context.Context*) – context instance
- **keypair** (*str*) – name of the key pair to create
- **ssm_parameter_name** (*str, optional*) – path to an SSM store parameter to receive the generated private key, instead of importing it or storing it locally.
- **ssm_key_id** (*str, optional*) – ID of a KMS key to encrypt the SSM parameter with. If omitted, the default key will be used.
- **public_key_path** (*str, optional*) – path to a public key file to be imported instead of generating a new key. Incompatible with the SSM options, as the private key will not be available for storing.

Returns

status (str): one of "exists", "imported" or "created" key_name (str): name of the key pair fingerprint (str): fingerprint of the key pair file_path (str, optional): if a new key was created, the path to

the file where the private key was stored

Return type In case of failure `False`, otherwise a dict containing

stacker.hooks.keypair.**get_existing_key_pair**(*ec2*, *keypair_name*)

stacker.hooks.keypair.**import_key_pair**(*ec2*, *keypair_name*, *public_key_data*)

stacker.hooks.keypair.**interactive_prompt**(*keypair_name*)

stacker.hooks.keypair.**read_public_key_file**(*path*)

stacker.hooks.route53 module

stacker.hooks.route53.**create_domain**(*provider*, *context*, ***kwargs*)

Create a domain within route53.

Parameters

- **provider** (*stacker.providers.base.BaseProvider*) – provider instance

- **context** (*stacker.context.Context*) – context instance

Returns: boolean for whether or not the hook succeeded.

stacker.hooks.utils module

stacker.hooks.utils.**full_path**(*path*)

stacker.hooks.utils.**handle_hooks**(*stage*, *hooks*, *provider*, *context*)
Used to handle pre/post_build hooks.

These are pieces of code that we want to run before/after the builder builds the stacks.

Parameters

- **stage** (*string*) – The current stage (pre_run, post_run, etc).
- **hooks** (*list*) – A list of *stacker.config.Hook* containing the hooks to execute.
- **provider** (stacker.provider.base.BaseProvider) – The provider the current stack is using.
- **context** (*stacker.context.Context*) – The current stacker context.

Module contents

stacker.logger package

Submodules

stacker.logger.formatter module

stacker.logger.handler module

Module contents

class stacker.logger.**ColorFormatter**(*fmt=None*, *datefmt=None*)
Bases: logging.Formatter

Handles colorizing formatted log messages if color provided.

format(*record*)
Format the specified record as text.

The record's attribute dictionary is used as the operand to a string formatting operation which yields the returned string. Before formatting the dictionary, a couple of preparatory steps are carried out. The message attribute of the record is computed using LogRecord.getMessage(). If the formatting string uses the time (as determined by a call to usesTime(), formatTime() is called to format the event time. If there is exception information, it is formatted using formatException() and appended to the message.

stacker.logger.**setup_logging**(*verbosity*, *formats=None*)
Configure a proper logger based on verbosity and optional log formats.

Parameters

- **verbosity** (*int*) – 0, 1, 2

- **formats** (*dict*) – Optional, looks for *info*, *color*, and *debug* keys which may override the associated default log formats.

stacker.lookups package

Subpackages

stacker.lookups.handlers package

Submodules

stacker.lookups.handlers.ami module

class stacker.lookups.handlers.ami.**AmiLookup**
Bases: *stacker.lookups.handlers.LookupHandler*

classmethod handle(*value*, *provider*, ***kwargs*)
Fetch the most recent AMI Id using a filter

For example:

${ami [<region>@]owners:self,account,amazon name_regex:serverX-[0-9]+ architecture:x64,i386}

The above fetches the most recent AMI where owner is self account or amazon and the ami name matches the regex described, the architecture will be either x64 or i386

You can also optionally specify the region in which to perform the AMI lookup.

Valid arguments:

owners (comma delimited) REQUIRED ONCE: aws_account_id | amazon | self

name_regex (a regex) REQUIRED ONCE: e.g. my-ubuntu-server-[0-9]+

executable_users (comma delimited) OPTIONAL ONCE: aws_account_id | amazon | self

Any other arguments specified are sent as filters to the aws api For example, "architecture:x86_64" will add a filter

exception stacker.lookups.handlers.ami.**ImageNotFound**(*search_string*)
Bases: exceptions.Exception

stacker.lookups.handlers.default module

class stacker.lookups.handlers.default.**DefaultLookup**
Bases: *stacker.lookups.handlers.LookupHandler*

classmethod handle(*value*, ***kwargs*)

Use a value from the environment or fall back to a default if the environment doesn't contain the variable.

Format of value:

<env_var>::<default value>

For example:

Groups: ${default app_security_groups::sg-12345,sg-67890}

If *app_security_groups* is defined in the environment, its defined value will be returned. Otherwise, *sg-12345,sg-67890* will be the returned value.

This allows defaults to be set at the config file level.

stacker.lookups.handlers.dynamodb module

class stacker.lookups.handlers.dynamodb.**DynamodbLookup**
Bases: *stacker.lookups.handlers.LookupHandler*

classmethod handle(*value*, ***kwargs*)
Get a value from a dynamodb table

dynamodb field types should be in the following format:

[<region>:]<tablename>@<primarypartionkey>:<keyvalue>.<keyvalue>...

Note: The region is optional, and defaults to the environment's *AWS_DEFAULT_REGION* if not specified.

stacker.lookups.handlers.envvar module

class stacker.lookups.handlers.envvar.**EnvvarLookup**
Bases: *stacker.lookups.handlers.LookupHandler*

classmethod handle(*value*, ***kwargs*)
Retrieve an environment variable.

For example:

In stacker we would reference the environment variable like this: conf_key: ${envvar ENV_VAR_NAME}

You can optionally store the value in a file, ie:

$ cat envvar_value.txt ENV_VAR_NAME

and reference it within stacker (NOTE: the path should be relative to the stacker config file):

conf_key: ${envvar file://envvar_value.txt}

Both of the above would resolve to conf_key: ENV_VALUE

stacker.lookups.handlers.file module

class stacker.lookups.handlers.file.**FileLookup**
Bases: *stacker.lookups.handlers.LookupHandler*

classmethod handle(*value*, ***kwargs*)
Translate a filename into the file contents.

Fields should use the following format:

```
<codec>:<path>
```

For example:

```
# We've written a file to /some/path:
$ echo "hello there" > /some/path

# In stacker we would reference the contents of this file with the
# following
conf_key: ${file plain:file://some/path}

# The above would resolve to
conf_key: hello there

# Or, if we used wanted a base64 encoded copy of the file data
conf_key: ${file base64:file://some/path}

# The above would resolve to
conf_key: aGVsbG8gdGhlcmUK
```

Supported codecs:

- plain
- base64 - encode the plain text file at the given path with base64 prior to returning it
- parameterized - the same as plain, but additionally supports referencing template parameters to create userdata that's supplemented with information from the template, as is commonly needed in EC2 UserData. For example, given a template parameter of BucketName, the file could contain the following text:

  ```
  #!/bin/sh
  aws s3 sync s3://{{BucketName}}/somepath /somepath
  ```

  and then you could use something like this in the YAML config file:

  ```
  UserData: ${file parameterized:/path/to/file}
  ```

  resulting in the UserData parameter being defined as:

  ```
  { "Fn::Join" : ["", [
      "#!/bin/sh\naws s3 sync s3://",
      {"Ref" : "BucketName"},
      "/somepath /somepath"
  ]] }
  ```

- parameterized-b64 - the same as parameterized, with the results additionally wrapped in *{ "Fn::Base64": ... }* , which is what you actually need for EC2 UserData

When using parameterized-b64 for UserData, you should use a variable defined as such:

```
from troposphere import AWSHelperFn

  "UserData": {
      "type": AWSHelperFn,
      "description": "Instance user data",
      "default": Ref("AWS::NoValue")
  }
```

and then assign UserData in a LaunchConfiguration or Instance to *self.get_variables()["UserData"]*. Note that we use AWSHelperFn as the type because the parameterized-b64 codec returns either a Base64 or a GenericHelperFn troposphere object

class stacker.lookups.handlers.file.**SafeUnicodeLoader**(*stream*)
Bases: yaml.loader.SafeLoader

construct_yaml_str(*node*)

stacker.lookups.handlers.file.**json_codec**(*raw*, *parameterized=False*)

stacker.lookups.handlers.file.**parameterized_codec**(*raw*, *b64*)
Parameterize a string, possibly encoding it as Base64 afterwards

Parameters

- **raw** (*str | bytes*) – String to be processed. Byte strings will be interpreted as UTF-8.
- **b64** (*bool*) – Whether to wrap the output in a Base64 CloudFormation call

Returns output to be included in a CloudFormation template.

Return type troposphere.AWSHelperFn

stacker.lookups.handlers.file.**yaml_codec**(*raw*, *parameterized=False*)

stacker.lookups.handlers.hook_data module

class stacker.lookups.handlers.hook_data.**HookDataLookup**
Bases: *stacker.lookups.handlers.LookupHandler*

classmethod handle(*value*, *context*, ***kwargs*)
Returns the value of a key for a given hook in hook_data.

Format of value:

<hook_name>::<key>

stacker.lookups.handlers.kms module

class stacker.lookups.handlers.kms.**KmsLookup**
Bases: *stacker.lookups.handlers.LookupHandler*

classmethod handle(*value*, ***kwargs*)
Decrypt the specified value with a master key in KMS.

kmssimple field types should be in the following format:

[<region>@]<base64 encrypted value>

Note: The region is optional, and defaults to the environment's *AWS_DEFAULT_REGION* if not specified.

For example:

We use the aws cli to get the encrypted value for the string # "PASSWORD" using the master key called "myStackerKey" in # us-east-1 $ aws –region us-east-1 kms encrypt –key-id alias/myStackerKey –plaintext "PASSWORD" –output text –query CiphertextBlob

CiD6bC8t2Y<...encrypted blob...>

In stacker we would reference the encrypted value like: conf_key: ${kms us-east-1@CiD6bC8t2Y<...encrypted blob...>}

You can optionally store the encrypted value in a file, ie:

kms_value.txt us-east-1@CiD6bC8t2Y<...encrypted blob...>

and reference it within stacker (NOTE: the path should be relative to the stacker config file):

conf_key: ${kms file://kms_value.txt}

Both of the above would resolve to conf_key: PASSWORD

stacker.lookups.handlers.output module

class stacker.lookups.handlers.output.**Output**(*stack_name*, *output_name*)
Bases: tuple

output_name
Alias for field number 1

stack_name
Alias for field number 0

class stacker.lookups.handlers.output.**OutputLookup**
Bases: *stacker.lookups.handlers.LookupHandler*

classmethod dependencies(*lookup_data*)
Calculate any dependencies required to perform this lookup.

Note that lookup_data may not be (completely) resolved at this time.

Parameters **lookup_data** – Parameter(s) given to this lookup

:type lookup_data VariableValue :return: Set of stack names (str) this lookup depends on :rtype: set

classmethod handle(*value*, *context=None*, ***kwargs*)
Fetch an output from the designated stack.

Parameters

- **value** (*str*) – string with the following format: <stack_name>::<output_name>, ie. some-stack::SomeOutput
- **context** (*stacker.context.Context*) – stacker context

Returns output from the specified stack

Return type str

stacker.lookups.handlers.output.**deconstruct**(*value*)

stacker.lookups.handlers.rxref module

Handler for fetching outputs from fully qualified stacks.

The *output* handler supports fetching outputs from stacks created within a sigle config file. Sometimes it's useful to fetch outputs from stacks created outside of the current config file. *rxref* supports this by not using the *stacker.context.Context* to expand the fqn of the stack.

Example

conf_value: ${rxref some-relative-fully-qualified-stack-name::SomeOutputName}

class stacker.lookups.handlers.rxref.**RxrefLookup**
Bases: *stacker.lookups.handlers.LookupHandler*

classmethod handle(*value*, *provider=None*, *context=None*, ***kwargs*)
Fetch an output from the designated stack.

Parameters

- **value** (*str*) – string with the following format: <stack_name>::<output_name>, ie. some-stack::SomeOutput
- **provider** (stacker.provider.base.BaseProvider) – subclass of the base provider
- **context** (*stacker.context.Context*) – stacker context

Returns output from the specified stack

Return type str

stacker.lookups.handlers.split module

class stacker.lookups.handlers.split.**SplitLookup**
Bases: *stacker.lookups.handlers.LookupHandler*

classmethod handle(*value*, ***kwargs*)
Split the supplied string on the given delimiter, providing a list.

Format of value:

<delimiter>::<value>

For example:

Subnets: ${split ,::subnet-1,subnet-2,subnet-3}

Would result in the variable *Subnets* getting a list consisting of:

["subnet-1", "subnet-2", "subnet-3"]

This is particularly useful when getting an output from another stack that contains a list. For example, the standard vpc blueprint outputs the list of Subnets it creates as a pair of Outputs (PublicSubnets, PrivateSubnets) that are comma separated, so you could use this in your config:

Subnets: ${split ,::${output vpc::PrivateSubnets}}

stacker.lookups.handlers.ssmstore module

class stacker.lookups.handlers.ssmstore.**SsmstoreLookup**
Bases: *stacker.lookups.handlers.LookupHandler*

classmethod handle(*value*, ***kwargs*)
Retrieve (and decrypt if applicable) a parameter from AWS SSM Parameter Store.

ssmstore field types should be in the following format:

[<region>@]ssmkey

Note: The region is optional, and defaults to us-east-1 if not given.

For example:

In stacker we would reference the encrypted value like: conf_key: ${ssmstore us-east-1@ssmkey}

You can optionally store the value in a file, ie:

ssmstore_value.txt us-east-1@ssmkey

and reference it within stacker (NOTE: the path should be relative to the stacker config file):

conf_key: ${ssmstore file://ssmstore_value.txt}

Both of the above would resolve to conf_key: PASSWORD

stacker.lookups.handlers.xref module

Handler for fetching outputs from fully qualified stacks.

The *output* handler supports fetching outputs from stacks created within a sigle config file. Sometimes it's useful to fetch outputs from stacks created outside of the current config file. *xref* supports this by not using the *stacker.context.Context* to expand the fqn of the stack.

Example

conf_value: ${xref some-fully-qualified-stack-name::SomeOutputName}

class stacker.lookups.handlers.xref.**XrefLookup**
Bases: *stacker.lookups.handlers.LookupHandler*

classmethod handle(*value*, *provider=None*, ***kwargs*)
Fetch an output from the designated stack.

Parameters

- **value** (*str*) – string with the following format: <stack_name>::<output_name>, ie. some-stack::SomeOutput
- **provider** (stacker.provider.base.BaseProvider) – subclass of the base provider

Returns output from the specified stack

Return type str

Module contents

class stacker.lookups.handlers.**LookupHandler**
Bases: object

classmethod dependencies(*lookup_data*)
Calculate any dependencies required to perform this lookup.

Note that lookup_data may not be (completely) resolved at this time.

Parameters **lookup_data** – Parameter(s) given to this lookup

:type lookup_data VariableValue :return: Set of stack names (str) this lookup depends on :rtype: set

classmethod handle(*value*, *context*, *provider*)
Perform the actual lookup

Parameters

- **value** (*str*) – Parameter(s) given to this lookup
- **context** –

- **provider** –

Returns Looked-up value

Return type str

Submodules

stacker.lookups.registry module

stacker.lookups.registry.**register_lookup_handler**(*lookup_type*, *handler_or_path*)
Register a lookup handler.

Parameters

- **lookup_type** (*str*) – Name to register the handler under
- **handler_or_path** (*OneOf[func, str]*) – a function or a path to a handler

stacker.lookups.registry.**resolve_lookups**(*variable*, *context*, *provider*)
Resolve a set of lookups.

Parameters

- **variable** (*stacker.variables.Variable*) – The variable resolving it's lookups.
- **context** (*stacker.context.Context*) – stacker context
- **provider** (stacker.provider.base.BaseProvider) – subclass of the base provider

Returns dict of Lookup -> resolved value

Return type dict

stacker.lookups.registry.**unregister_lookup_handler**(*lookup_type*)
Unregister the specified lookup type.

This is useful when testing various lookup types if you want to unregister the lookup type after the test runs.

Parameters **lookup_type** (*str*) – Name of the lookup type to unregister

Module contents

class stacker.lookups.**Lookup**(*type*, *input*, *raw*)
Bases: tuple

input
Alias for field number 1

raw
Alias for field number 2

type
Alias for field number 0

stacker.lookups.**extract_lookups**(*value*)
Recursively extracts any stack lookups within the data structure.

Parameters **value** (*one of str, list, dict*) – a structure that contains lookups to output values

Returns list of lookups if any

Return type list

stacker.lookups.**extract_lookups_from_string**(*value*)
Extract any lookups within a string.

Parameters **value** (*str*) – string value we're extracting lookups from

Returns list of *stacker.lookups.Lookup* if any

Return type list

stacker.providers package

Subpackages

stacker.providers.aws package

Submodules

stacker.providers.aws.default module

class stacker.providers.aws.default.**Provider**(*session*, *region=None*, *interactive=False*, *replacements_only=False*, *recreate_failed=False*, *service_role=None*, ***kwargs*)
Bases: *stacker.providers.base.BaseProvider*

AWS CloudFormation Provider

COMPLETE_STATUSES = ('CREATE_COMPLETE', 'DELETE_COMPLETE', 'IMPORT_COMPLETE', 'UPDATE_

DELETED_STATUS = 'DELETE_COMPLETE'

FAILED_STATUSES = ('CREATE_FAILED', 'ROLLBACK_FAILED', 'ROLLBACK_COMPLETE', 'DELETE_FA

IN_PROGRESS_STATUSES = ('CREATE_IN_PROGRESS', 'IMPORT_IN_PROGRESS', 'UPDATE_IN_PROGRES

RECREATION_STATUSES = ('CREATE_FAILED', 'ROLLBACK_FAILED', 'ROLLBACK_COMPLETE')

REVIEW_STATUS = 'REVIEW_IN_PROGRESS'

ROLLING_BACK_STATUSES = ('ROLLBACK_IN_PROGRESS', 'IMPORT_ROLLBACK_IN_PROGRESS', 'UPDAT

create_stack(*fqn*, *template*, *parameters*, *tags*, *force_change_set=False*, *stack_policy=None*, *notification_arns=None*, ***kwargs*)
Create a new Cloudformation stack.

Parameters

- **fqn** (*str*) – The fully qualified name of the Cloudformation stack.
- **template** (*stacker.providers.base.Template*) – A Template object to use when creating the stack.
- **parameters** (*list*) – A list of dictionaries that defines the parameter list to be applied to the Cloudformation stack.
- **tags** (*list*) – A list of dictionaries that defines the tags that should be applied to the Cloudformation stack.

- **force_change_set** (*bool*) – Whether or not to force change set use.
- **stack_policy** (*stacker.providers.base.Template*) – A template object representing a stack policy.
- **notification_arns** (*list, optional*) – An optional list of SNS topic ARNs to send CloudFormation Events to.

deal_with_changeset_stack_policy(*fqn*, *stack_policy*)
Set a stack policy when using changesets.

ChangeSets don't allow you to set stack policies in the same call to update them. This sets it before executing the changeset if the stack policy is passed in.

Parameters **stack_policy** (*stacker.providers.base.Template*) – A template object representing a stack policy.

default_update_stack(*fqn*, *template*, *old_parameters*, *parameters*, *tags*, *stack_policy=None*, *notification_arns=[]*, ***kwargs*)
Update a Cloudformation stack in default mode.

Parameters

- **fqn** (*str*) – The fully qualified name of the Cloudformation stack.
- **template** (*stacker.providers.base.Template*) – A Template object to use when updating the stack.
- **old_parameters** (*list*) – A list of dictionaries that defines the parameter list on the existing Cloudformation stack.
- **parameters** (*list*) – A list of dictionaries that defines the parameter list to be applied to the Cloudformation stack.
- **tags** (*list*) – A list of dictionaries that defines the tags that should be applied to the Cloudformation stack.
- **stack_policy** (*stacker.providers.base.Template*) – A template object representing a stack policy.

destroy_stack(*stack*, ***kwargs*)

get_events(*stack_name*, *chronological=True*)
Get the events in batches and return in chronological order

get_output_dict(*stack*)

get_outputs(*stack_name*, **args*, ***kwargs*)

get_rollback_status_reason(*stack_name*)
Process events and returns latest roll back reason

get_stack(*stack_name*, ***kwargs*)

get_stack_changes(*stack*, *template*, *parameters*, *tags*, ***kwargs*)
Get the changes from a ChangeSet.

Parameters

- **stack** (*stacker.stack.Stack*) – the stack to get changes
- **template** (*stacker.providers.base.Template*) – A Template object to compaired to.
- **parameters** (*list*) – A list of dictionaries that defines the parameter list to be applied to the Cloudformation stack.

- **tags** (*list*) – A list of dictionaries that defines the tags that should be applied to the Cloudformation stack.

Returns Stack outputs with inferred changes.

Return type dict

get_stack_info(*stack*)
Get the template and parameters of the stack currently in AWS

Returns [template, parameters]

get_stack_name(*stack*, ***kwargs*)

get_stack_status(*stack*, ***kwargs*)

get_stack_tags(*stack*, ***kwargs*)

interactive_update_stack(*fqn*, *template*, *old_parameters*, *parameters*, *stack_policy*, *tags*, ***kwargs*)
Update a Cloudformation stack in interactive mode.

Parameters

- **fqn** (*str*) – The fully qualified name of the Cloudformation stack.
- **template** (*stacker.providers.base.Template*) – A Template object to use when updating the stack.
- **old_parameters** (*list*) – A list of dictionaries that defines the parameter list on the existing Cloudformation stack.
- **parameters** (*list*) – A list of dictionaries that defines the parameter list to be applied to the Cloudformation stack.
- **stack_policy** (*stacker.providers.base.Template*) – A template object representing a stack policy.
- **tags** (*list*) – A list of dictionaries that defines the tags that should be applied to the Cloudformation stack.

is_stack_completed(*stack*, ***kwargs*)

is_stack_destroyed(*stack*, ***kwargs*)

is_stack_failed(*stack*, ***kwargs*)

is_stack_in_progress(*stack*, ***kwargs*)

is_stack_in_review(*stack*, ***kwargs*)

is_stack_recreatable(*stack*, ***kwargs*)

is_stack_rolling_back(*stack*, ***kwargs*)

noninteractive_changeset_update(*fqn*, *template*, *old_parameters*, *parameters*, *stack_policy*, *tags*, ***kwargs*)
Update a Cloudformation stack using a change set.

This is required for stacks with a defined Transform (i.e. SAM), as the default update_stack API cannot be used with them.

Parameters

- **fqn** (*str*) – The fully qualified name of the Cloudformation stack.
- **template** (*stacker.providers.base.Template*) – A Template object to use when updating the stack.

- **old_parameters** (*list*) – A list of dictionaries that defines the parameter list on the existing Cloudformation stack.
- **parameters** (*list*) – A list of dictionaries that defines the parameter list to be applied to the Cloudformation stack.
- **stack_policy** (*stacker.providers.base.Template*) – A template object representing a stack policy.
- **tags** (*list*) – A list of dictionaries that defines the tags that should be applied to the Cloudformation stack.

static params_as_dict(*parameters_list*)

prepare_stack_for_update(*stack*, *tags*)
Prepare a stack for updating

It may involve deleting the stack if is has failed it's initial creation. The deletion is only allowed if:

- The stack contains all the tags configured in the current context;
- The stack is in one of the statuses considered safe to re-create
- `recreate_failed` is enabled, due to either being explicitly enabled by the user, or because interactive mode is on.

Parameters

- **stack** (*dict*) – a stack object returned from get_stack
- **tags** (*list*) – list of expected tags that must be present in the stack if it must be re-created

Returns

True if the stack can be updated, False if it must be re-created

Return type bool

select_update_method(*force_interactive*, *force_change_set*)
Select the correct update method when updating a stack.

Parameters

- **force_interactive** (*str*) – Whether or not to force interactive mode no matter what mode the provider is in.
- **force_change_set** (*bool*) – Whether or not to force change set use.

Returns The correct object method to use when updating.

Return type function

tail(*stack_name*, *cancel*, *log_func=<staticmethod object>*, *sleep_time=5*, *include_initial=True*)
Show and then tail the event log

tail_stack(*stack*, *cancel*, *log_func=None*, ***kwargs*)

update_stack(*fqn*, *template*, *old_parameters*, *parameters*, *tags*, *force_interactive=False*, *force_change_set=False*, *stack_policy=None*, ***kwargs*)
Update a Cloudformation stack.

Parameters

- **fqn** (*str*) – The fully qualified name of the Cloudformation stack.

- **template** (*stacker.providers.base.Template*) – A Template object to use when updating the stack.
- **old_parameters** (*list*) – A list of dictionaries that defines the parameter list on the existing Cloudformation stack.
- **parameters** (*list*) – A list of dictionaries that defines the parameter list to be applied to the Cloudformation stack.
- **tags** (*list*) – A list of dictionaries that defines the tags that should be applied to the Cloudformation stack.
- **force_interactive** (*bool*) – A flag that indicates whether the update should be interactive. If set to True, interactive mode will be used no matter if the provider is in interactive mode or not. False will follow the behavior of the provider.
- **force_change_set** (*bool*) – A flag that indicates whether the update must be executed with a change set.
- **stack_policy** (*stacker.providers.base.Template*) – A template object representing a stack policy.

class stacker.providers.aws.default.**ProviderBuilder**(*region=None*, ***kwargs*)

Bases: future.types.newobject.newobject

Implements a Memoized ProviderBuilder for the AWS provider.

build(*region=None*, *profile=None*)

Get or create the provider for the given region and profile.

stacker.providers.aws.default.**ask_for_approval**(*full_changeset=None*, *params_diff=None*, *include_verbose=False*, *fqn=None*)

Prompt the user for approval to execute a change set.

Parameters

- **full_changeset** (*list, optional*) – A list of the full changeset that will be output if the user specifies verbose.
- **params_diff** (*list, optional*) – A list of DictValue detailing the differences between two parameters returned by *stacker.actions.diff.diff_dictionaries()*
- **include_verbose** (*bool, optional*) – Boolean for whether or not to include the verbose option.
- **fqn** (*str*) – fully qualified name of the stack.

stacker.providers.aws.default.**check_tags_contain**(*actual*, *expected*)

Check if a set of AWS resource tags is contained in another

Every tag key in *expected* must be present in *actual*, and have the same value. Extra keys in *actual* but not in *expected* are ignored.

Parameters

- **actual** (*list*) – Set of tags to be verified, usually from the description of a resource. Each item must be a *dict* containing *Key* and *Value* items.
- **expected** (*list*) – Set of tags that must be present in *actual* (in the same format).

stacker.providers.aws.default.**create_change_set**(*cfn_client*, *fqn*, *template*, *parameters*, *tags*, *change_set_type='UPDATE'*, *replacements_only=False*, *service_role=None*, *notification_arns=None*)

stacker.providers.aws.default.**format_params_diff**(*params_diff*)

Just a wrapper for stacker.actions.diff.format_params_diff for testing purposes.

stacker.providers.aws.default.**generate_cloudformation_args**(*stack_name*, *parameters*, *tags*, *template*, *capabilities=['CAPABILITY_NAMED_IAM', 'CAPABILITY_AUTO_EXPAND']*, *change_set_type=None*, *service_role=None*, *stack_policy=None*, *change_set_name=None*, *notification_arns=None*)

Used to generate the args for common cloudformation API interactions.

This is used for create_stack/update_stack/create_change_set calls in cloudformation.

Parameters

- **stack_name** (*str*) – The fully qualified stack name in Cloudformation.
- **parameters** (*list*) – A list of dictionaries that defines the parameter list to be applied to the Cloudformation stack.
- **tags** (*list*) – A list of dictionaries that defines the tags that should be applied to the Cloudformation stack.
- **template** (stacker.provider.base.Template) – The template object.
- **capabilities** (*list, optional*) – A list of capabilities to use when updating Cloudformation.
- **change_set_type** (*str, optional*) – An optional change set type to use with create_change_set.
- **service_role** (*str, optional*) – An optional service role to use when interacting with Cloudformation.
- **stack_policy** (*stacker.providers.base.Template*) – A template object representing a stack policy.
- **change_set_name** (*str, optional*) – An optional change set name to use with create_change_set.
- **notification_arns** (*list, optional*) – An optional list of SNS topic ARNs to send CloudFormation Events to.

Returns

A dictionary of arguments to be used in the Cloudformation API call.

Return type dict

stacker.providers.aws.default.**generate_stack_policy_args**(*stack_policy=None*)

Converts a stack policy object into keyword args.

Parameters **stack_policy** (*stacker.providers.base.Template*) – A template object representing a stack policy.

Returns A dictionary of keyword arguments to be used elsewhere.

Return type dict

stacker.providers.aws.default.**get_change_set_name**()
Return a valid Change Set Name.

The name has to satisfy the following regex: [a-zA-Z][-a-zA-Z0-9]*

And must be unique across all change sets.

stacker.providers.aws.default.**get_cloudformation_client**(*session*)

stacker.providers.aws.default.**get_output_dict**(*stack*)
Returns a dict of key/values for the outputs for a given CF stack.

Parameters **stack** (*dict*) – The stack object to get outputs from.

Returns A dictionary with key/values for each output on the stack.

Return type dict

stacker.providers.aws.default.**output_full_changeset**(*full_changeset=None*, *params_diff=None*, *answer=None*, *fqn=None*)
Optionally output full changeset.

Parameters

- **full_changeset** (*list, optional*) – A list of the full changeset that will be output if the user specifies verbose.
- **params_diff** (*list, optional*) – A list of DictValue detailing the differences between two parameters returned by *stacker.actions.diff.diff_dictionaries()*
- **answer** (*str, optional*) – predetermined answer to the prompt if it has already been answered or inferred.
- **fqn** (*str, optional*) – fully qualified name of the stack.

stacker.providers.aws.default.**output_summary**(*fqn*, *action*, *changeset*, *params_diff*, *replacements_only=False*)
Log a summary of the changeset.

Parameters

- **fqn** (*string*) – fully qualified name of the stack
- **action** (*string*) – action to include in the log message
- **changeset** (*list*) – AWS changeset
- **params_diff** (*list*) – A list of dictionaries detailing the differences between two parameters returned by *stacker.actions.diff.diff_dictionaries()*
- **replacements_only** (*bool, optional*) – boolean for whether or not we only want to list replacements

stacker.providers.aws.default.**requires_replacement**(*changeset*)
Return the changes within the changeset that require replacement.

Parameters **changeset** (*list*) – List of changes

Returns A list of changes that require replacement, if any.

Return type list

stacker.providers.aws.default.**s3_fallback**(*fqn*, *template*, *parameters*, *tags*, *method*, *change_set_name=None*, *service_role=None*)

stacker.providers.aws.default.**summarize_params_diff**(*params_diff*)

stacker.providers.aws.default.**wait_till_change_set_complete**(*cfn_client*, *change_set_id*, *try_count=25*, *sleep_time=0.5*, *max_sleep=3*)

Checks state of a changeset, returning when it is in a complete state.

Since changesets can take a little bit of time to get into a complete state, we need to poll it until it does so. This will try to get the state *try_count* times, waiting *sleep_time* * 2 seconds between each try up to the *max_sleep* number of seconds. If, after that time, the changeset is not in a complete state it fails. These default settings will wait a little over one minute.

Parameters

- **cfn_client** (botocore.client.CloudFormation) – Used to query cloudformation.
- **change_set_id** (*str*) – The unique changeset id to wait for.
- **try_count** (*int*) – Number of times to try the call.
- **sleep_time** (*int*) – Time to sleep between attempts.
- **max_sleep** (*int*) – Max time to sleep during backoff

Returns

The response from cloudformation for the describe_change_set call.

Return type dict

Module contents

Submodules

stacker.providers.base module

class stacker.providers.base.**BaseProvider**

Bases: future.types.newobject.newobject

create_stack(**args*, ***kwargs*)

destroy_stack(**args*, ***kwargs*)

get_output(*stack_name*, *output*)

get_outputs(*stack_name*, **args*, ***kwargs*)

get_stack(*stack_name*, **args*, ***kwargs*)

get_stack_status(*stack_name*, **args*, ***kwargs*)

update_stack(**args*, ***kwargs*)

class stacker.providers.base.**BaseProviderBuilder**
Bases: future.types.newobject.newobject

build(*region=None*)

class stacker.providers.base.**Template**(*url=None*, *body=None*)
Bases: future.types.newobject.newobject

A value object that represents a CloudFormation stack template, which could be optionally uploaded to s3.

Presence of the url attribute indicates that the template was uploaded to S3, and the uploaded template should be used for CreateStack/UpdateStack calls.

stacker.providers.base.**not_implemented**(*method*)

Module contents

Submodules

stacker.context module

class stacker.context.**Context**(*environment=None*, *stack_names=None*, *config=None*, *force_stacks=None*)
Bases: future.types.newobject.newobject

The context under which the current stacks are being executed.

The stacker Context is responsible for translating the values passed in via the command line and specified in the config to *Stack* objects.

Parameters

- **environment** (*dict*) – A dictionary used to pass in information about the environment. Useful for templating.
- **stack_names** (*list*) – A list of stack_names to operate on. If not passed, usually all stacks defined in the config will be operated on.
- **config** (*stacker.config.Config*) – The stacker configuration being operated on.
- **force_stacks** (*list*) – A list of stacks to force work on. Used to work on locked stacks.

bucket_name

get_fqn(*name=None*)
Return the fully qualified name of an object within this context.

If the name passed already appears to be a fully qualified name, it will be returned with no further processing.

get_stack(*name*)

get_stacks()
Get the stacks for the current action.

Handles configuring the *stacker.stack.Stack* objects that will be used in the current action.

Returns a list of *stacker.stack.Stack* objects

Return type list

get_stacks_dict()

get_targets()
Returns the named targets that are specified in the config.

Returns a list of `stacker.target.Target` objects

Return type list

mappings

namespace

namespace_delimiter

set_hook_data(*key*, *data*)
Set hook data for the given key.

Parameters

- **key** (*str*) – The key to store the hook data in.
- **data** (`collections.Mapping`) – A dictionary of data to store, as returned from a hook.

tags

template_indent

upload_templates_to_s3

`stacker.context.`**get_fqn**(*base_fqn*, *delimiter*, *name=None*)
Return the fully qualified name of an object within this context.

If the name passed already appears to be a fully qualified name, it will be returned with no further processing.

stacker.environment module

class `stacker.environment.`**DictWithSourceType**(*source_type*, **args*)
Bases: `dict`

An environment dict which keeps track of its source.

Environment files may be loaded from simple key/value files, or from structured YAML files, and we need to render them using a different strategy based on their source. This class adds a source_type property to a dict which keeps track of whether the source for the dict is yaml or simple.

`stacker.environment.`**parse_environment**(*raw_environment*)

`stacker.environment.`**parse_yaml_environment**(*raw_environment*)

stacker.exceptions module

exception `stacker.exceptions.`**CancelExecution**
Bases: `exceptions.Exception`

Exception raised when we want to cancel executing the plan.

exception `stacker.exceptions.`**ChangesetDidNotStabilize**(*change_set_id*)
Bases: `exceptions.Exception`

exception `stacker.exceptions.`**FailedLookup**(*lookup*, *error*, **args*, ***kwargs*)
Bases: `exceptions.Exception`

Intermediary Exception to be converted to FailedVariableLookup once it bubbles up there

exception stacker.exceptions.**FailedVariableLookup**(*variable_name*, *lookup*, *error*, **args*, ***kwargs*)
Bases: exceptions.Exception

exception stacker.exceptions.**GraphError**(*exception*, *stack*, *dependency*)
Bases: exceptions.Exception

Raised when the graph is invalid (e.g. acyclic dependencies)

exception stacker.exceptions.**ImproperlyConfigured**(*cls*, *error*, **args*, ***kwargs*)
Bases: exceptions.Exception

exception stacker.exceptions.**InvalidConfig**(*errors*)
Bases: exceptions.Exception

exception stacker.exceptions.**InvalidLookupCombination**(*lookup*, *lookups*, *value*, **args*, ***kwargs*)
Bases: exceptions.Exception

exception stacker.exceptions.**InvalidLookupConcatenation**(*lookup*, *lookups*, **args*, ***kwargs*)
Bases: exceptions.Exception

Intermediary Exception to be converted to InvalidLookupCombination once it bubbles up there

exception stacker.exceptions.**InvalidUserdataPlaceholder**(*blueprint_name*, *exception_message*, **args*, ***kwargs*)
Bases: exceptions.Exception

exception stacker.exceptions.**MissingEnvironment**(*key*, **args*, ***kwargs*)
Bases: exceptions.Exception

exception stacker.exceptions.**MissingParameterException**(*parameters*, **args*, ***kwargs*)
Bases: exceptions.Exception

exception stacker.exceptions.**MissingVariable**(*blueprint_name*, *variable_name*, **args*, ***kwargs*)
Bases: exceptions.Exception

exception stacker.exceptions.**OutputDoesNotExist**(*stack_name*, *output*, **args*, ***kwargs*)
Bases: exceptions.Exception

exception stacker.exceptions.**PlanFailed**(*failed_steps*, **args*, ***kwargs*)
Bases: exceptions.Exception

exception stacker.exceptions.**StackDidNotChange**
Bases: exceptions.Exception

Exception raised when there are no changes to be made by the provider.

exception stacker.exceptions.**StackDoesNotExist**(*stack_name*, **args*, ***kwargs*)
Bases: exceptions.Exception

exception stacker.exceptions.**StackUpdateBadStatus**(*stack_name*, *stack_status*, *reason*, **args*, ***kwargs*)
Bases: exceptions.Exception

exception stacker.exceptions.**UnableToExecuteChangeSet**(*stack_name*, *change_set_id*, *execution_status*)
Bases: exceptions.Exception

exception stacker.exceptions.**UnhandledChangeSetStatus**(*stack_name*, *change_set_id*, *status*, *status_reason*)
Bases: exceptions.Exception

exception stacker.exceptions.**UnknownLookupType**(*lookup_type*, **args*, ***kwargs*)
Bases: exceptions.Exception

exception stacker.exceptions.**UnresolvedVariable**(*blueprint_name*, *variable*, **args*, ***kwargs*)
Bases: exceptions.Exception

exception stacker.exceptions.**UnresolvedVariableValue**(*lookup*, **args*, ***kwargs*)
Bases: exceptions.Exception

Intermediary Exception to be converted to UnresolvedVariable once it bubbles up there

exception stacker.exceptions.**UnresolvedVariables**(*blueprint_name*, **args*, ***kwargs*)
Bases: exceptions.Exception

exception stacker.exceptions.**ValidatorError**(*variable*, *validator*, *value*, *exception=None*)
Bases: exceptions.Exception

Used for errors raised by custom validators of blueprint variables.

exception stacker.exceptions.**VariableTypeRequired**(*blueprint_name*, *variable_name*, **args*, ***kwargs*)
Bases: exceptions.Exception

exception stacker.exceptions.**WrongEnvironmentType**(*key*, **args*, ***kwargs*)
Bases: exceptions.Exception

stacker.plan module

class stacker.plan.**Graph**(*steps=None*, *dag=None*)
Bases: future.types.newobject.newobject

Graph represents a graph of steps.

The *Graph* helps organize the steps needed to execute a particular action for a set of *stacker.stack.Stack* objects. When initialized with a set of steps, it will first build a Directed Acyclic Graph from the steps and their dependencies.

Example:

```
>>> dag = DAG()
>>> a = Step("a", fn=build)
>>> b = Step("b", fn=build)
>>> dag.add_step(a)
>>> dag.add_step(b)
>>> dag.connect(a, b)
```

Parameters

- **steps** (*list*) – an optional list of *Step* objects to execute.
- **dag** (stacker.dag.DAG) – an optional stacker.dag.DAG object. If one is not provided, a new one will be initialized.

add_step(*step*)

connect(*step*, *dep*)

downstream(*step_name*)
Returns the direct dependencies of the given step

filtered(*step_names*)
Returns a "filtered" version of this graph.

to_dict()

topological_sort()

transitive_reduction()

transposed()
Returns a "transposed" version of this graph. Useful for walking in reverse.

walk(*walker*, *walk_func*)

class stacker.plan.**Plan**(*description*, *graph*)
Bases: future.types.newobject.newobject

A convenience class for working on a Graph. :param description: description of the plan. :type description: str :param graph: a graph of steps. :type graph: *Graph*

dump(*directory*, *context*, *provider=None*)

execute(**args*, ***kwargs*)
Walks each step in the underlying graph, and raises an exception if any of the steps fail.

Raises PlanFailed – Raised if any of the steps fail.

keys()

outline(*level=20*, *message=""*)
Print an outline of the actions the plan is going to take. The outline will represent the rough ordering of the steps that will be taken. :param level: a valid log level that should be used to log

the outline

Parameters **message** (*str, optional*) – a message that will be logged to the user after the outline has been logged.

step_names

steps

walk(*walker*)
Walks each step in the underlying graph, in topological order.

Parameters **walker** (*func*) – a walker function to be passed to stacker.dag.DAG to walk the graph.

class stacker.plan.**Step**(*stack*, *fn*, *watch_func=None*)
Bases: future.types.newobject.newobject

State machine for executing generic actions related to stacks. :param stack: the stack associated

with this step

Parameters

- **fn** (*func*) – the function to run to execute the step. This function will be ran multiple times until the step is "done".
- **watch_func** (*func*) – an optional function that will be called to "tail" the step action.

complete()
A shortcut for set_status(COMPLETE)

completed
Returns True if the step is in a COMPLETE state.

done
Returns True if the step is finished (either COMPLETE, SKIPPED or FAILED)

failed
Returns True if the step is in a FAILED state.

name

ok
Returns True if the step is finished (either COMPLETE or SKIPPED)

required_by

requires

run()
Runs this step until it has completed successfully, or been skipped.

set_status(*status*)
Sets the current step's status. :param status: The status to set the

step to.

skip()
A shortcut for set_status(SKIPPED)

skipped
Returns True if the step is in a SKIPPED state.

submit()
A shortcut for set_status(SUBMITTED)

submitted
Returns True if the step is SUBMITTED, COMPLETE, or SKIPPED.

stacker.plan.**build_graph**(*steps*)
Builds a graph of steps. :param steps: a list of *Step* objects to execute. :type steps: list

stacker.plan.**build_plan**(*description*, *graph*, *targets=None*, *reverse=False*)
Builds a plan from a list of steps. :param description: an arbitrary string to

describe the plan.

Parameters

- **graph** (*Graph*) – a list of *Graph* to execute.
- **targets** (*list*) – an optional list of step names to filter the graph to. If provided, only these steps, and their transitive dependencies will be executed. If no targets are specified, every node in the graph will be executed.
- **reverse** (*bool*) – If provided, the graph will be walked in reverse order (dependencies last).

stacker.plan.**log_step**(*step*)

stacker.session_cache module

stacker.session_cache.**get_session**(*region*, *profile=None*)
Creates a boto3 session with a cache

Parameters

- **region** (*str*) – The region for the session
- **profile** (*str*) – The profile for the session

Returns

A boto3 session with credential caching

Return type boto3.session.Session

stacker.stack module

class stacker.stack.**Stack**(*definition*, *context*, *variables=None*, *mappings=None*, *locked=False*, *force=False*, *enabled=True*, *protected=False*, *notification_arns=None*)
Bases: future.types.newobject.newobject

Represents gathered information about a stack to be built/updated.

Parameters

- **definition** (*stacker.config.Stack*) – A stack definition.
- **context** (*stacker.context.Context*) – Current context for building the stack.
- **mappings** (*dict, optional*) – Cloudformation mappings passed to the blueprint.
- **locked** (*bool, optional*) – Whether or not the stack is locked.
- **force** (*bool, optional*) – Whether to force updates on this stack.
- **enabled** (*bool, optional*) – Whether this stack is enabled.
- **protected** (*boot, optional*) – Whether this stack is protected.
- **notification_arns** (*list, optional*) – An optional list of SNS topic ARNs to send CloudFormation Events to.

all_parameter_definitions
Return a list of all parameters in the blueprint/template.

blueprint

parameter_values
Return all CloudFormation Parameters for the stack.

CloudFormation Parameters can be specified via Blueprint Variables with a *stacker.blueprints.variables.types.CFNType type*.

Returns dictionary of <parameter name>: <parameter value>.

Return type dict

required_by

required_parameter_definitions
Return all the required CloudFormation Parameters for the stack.

requires

resolve(*context*, *provider*)
Resolve the Stack variables.

This resolves the Stack variables and then prepares the Blueprint for rendering by passing the resolved variables to the Blueprint.

Parameters

- **context** (*stacker.context.Context*) – stacker context
- **provider** (stacker.provider.base.BaseProvider) – subclass of the base provider

set_outputs(*outputs*)

stack_policy

tags
Returns the tags that should be set on this stack. Includes both the global tags, as well as any stack specific tags or overrides.

Returns dictionary of tags

Return type dict

stacker.status module

class stacker.status.**CompleteStatus**(*reason=None*)
Bases: *stacker.status.Status*

class stacker.status.**DidNotChangeStatus**(*reason=None*)
Bases: *stacker.status.SkippedStatus*

reason = 'nochange'

class stacker.status.**FailedStatus**(*reason=None*)
Bases: *stacker.status.Status*

class stacker.status.**NotSubmittedStatus**(*reason=None*)
Bases: *stacker.status.SkippedStatus*

reason = 'disabled'

class stacker.status.**NotUpdatedStatus**(*reason=None*)
Bases: *stacker.status.SkippedStatus*

reason = 'locked'

class stacker.status.**PendingStatus**(*reason=None*)
Bases: *stacker.status.Status*

class stacker.status.**SkippedStatus**(*reason=None*)
Bases: *stacker.status.Status*

class stacker.status.**StackDoesNotExist**(*reason=None*)
Bases: *stacker.status.SkippedStatus*

reason = 'does not exist in cloudformation'

class stacker.status.**Status**(*name*, *code*, *reason=None*)
Bases: future.types.newobject.newobject

class stacker.status.**SubmittedStatus**(*reason=None*)
Bases: *stacker.status.Status*

stacker.tokenize_userdata module

stacker.tokenize_userdata.**cf_tokenize**(*s*)
Parses UserData for Cloudformation helper functions.

http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/user-data.html http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/intrinsic-function-reference.html http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/quickref-cloudformation.html#scenario-userdata-base64

It breaks apart the given string at each recognized function (see HELPERS) and instantiates the helper function objects in place of those.

Returns a list of parts as a result. Useful when used with Join() and Base64() CloudFormation functions to produce user data.

ie: Base64(Join('', cf_tokenize(userdata_string)))

stacker.util module

class stacker.util.**Extractor**(*archive=None*)
Bases: future.types.newobject.newobject

Base class for extractors.

static extension()
Serve as placeholder; override this in subclasses.

set_archive(*dir_name*)
Update archive filename to match directory name & extension.

Parameters **dir_name** (*string*) – Archive directory name

class stacker.util.**SOARecord**(*record*)
Bases: future.types.newobject.newobject

Represents an SOA record.

class stacker.util.**SOARecordText**(*record_text*)
Bases: future.types.newobject.newobject

Represents the actual body of an SOARecord.

class stacker.util.**SourceProcessor**(*sources*, *stacker_cache_dir=None*)
Bases: future.types.newobject.newobject

Makes remote python package sources available in current environment.

ISO8601_FORMAT = '%Y%m%dT%H%M%SZ'

create_cache_directories()
Ensure that SourceProcessor cache directories exist.

determine_git_ls_remote_ref(*config*)
Determine the ref to be used with the "git ls-remote" command.

Parameters **config** (*stacker.config.GitPackageSource*) – git config dictionary; 'branch' key is optional

Returns A branch reference or "HEAD"

Return type str

determine_git_ref(*config*)
Determine the ref to be used for 'git checkout'.

Parameters **config** (*dict*) – git config dictionary

Returns A commit id or tag name

Return type str

fetch_git_package(*config*)
Make a remote git repository available for local use.

Parameters **config** (*dict*) – git config dictionary

fetch_local_package(*config*)
Make a local path available to current stacker config.

Parameters **config** (*dict*) – 'local' path config dictionary

fetch_s3_package(*config*)
Make a remote S3 archive available for local use.

Parameters **config** (*dict*) – git config dictionary

get_package_sources()
Make remote python packages available for local use.

git_ls_remote(*uri*, *ref*)
Determine the latest commit id for a given ref.

Parameters

- **uri** (*string*) – git URI
- **ref** (*string*) – git ref

Returns A commit id

Return type str

sanitize_git_path(*uri*, *ref=None*)
Take a git URI and ref and converts it to a directory safe path.

Parameters

- **uri** (*string*) – git URI (e.g. git@github.com:foo/bar.git)
- **ref** (*string*) – optional git ref to be appended to the path

Returns Directory name for the supplied uri

Return type str

sanitize_uri_path(*uri*)
Take a URI and converts it to a directory safe path.

Parameters **uri** (*string*) – URI (e.g. http://example.com/cats)

Returns Directory name for the supplied uri

Return type str

update_paths_and_config(*config*, *pkg_dir_name*, *pkg_cache_dir=None*)
Handle remote source defined sys.paths & configs.

Parameters

- **config** (*dict*) – git config dictionary
- **pkg_dir_name** (*string*) – directory name of the stacker archive
- **pkg_cache_dir** (*string*) – fully qualified path to stacker cache cache directory

class stacker.util.**TarExtractor**(*archive=None*)
Bases: *stacker.util.Extractor*

Extracts tar archives.

static extension()
Return archive extension.

extract(*destination*)
Extract the archive.

class stacker.util.**TarGzipExtractor**(*archive=None*)
Bases: *stacker.util.Extractor*

Extracts compressed tar archives.

static extension()
Return archive extension.

extract(*destination*)
Extract the archive.

class stacker.util.**ZipExtractor**(*archive=None*)
Bases: *stacker.util.Extractor*

Extracts zip archives.

static extension()
Return archive extension.

extract(*destination*)
Extract the archive.

stacker.util.**camel_to_snake**(*name*)
Converts CamelCase to snake_case.

Parameters **name** (*string*) – The name to convert from CamelCase to snake_case.

Returns Converted string.

Return type string

stacker.util.**cf_safe_name**(*name*)
Converts a name to a safe string for a Cloudformation resource.

Given a string, returns a name that is safe for use as a CloudFormation Resource. (ie: Only alphanumeric characters)

stacker.util.**convert_class_name**(*kls*)
Gets a string that represents a given class.

Parameters **kls** (*class*) – The class being analyzed for its name.

Returns The name of the given kls.

Return type string

stacker.util.**create_route53_zone**(*client*, *zone_name*)
Creates the given zone_name if it doesn't already exists.

Also sets the SOA negative caching TTL to something short (300 seconds).

Parameters

- **client** (botocore.client.Route53) – The connection used to interact with Route53's API.

- **zone_name** (*string*) – The name of the DNS hosted zone to create.

Returns

The zone id returned from AWS for the existing, or newly created zone.

Return type string

stacker.util.**ensure_s3_bucket**(*s3_client*, *bucket_name*, *bucket_region*)
Ensure an s3 bucket exists, if it does not then create it.

Parameters

- **s3_client** (botocore.client.Client) – An s3 client used to verify and create the bucket.
- **bucket_name** (*str*) – The bucket being checked/created.
- **bucket_region** (*str, optional*) – The region to create the bucket in. If not provided, will be determined by s3_client's region.

stacker.util.**get_client_region**(*client*)
Gets the region from a boto3.client.Client object.

Parameters **client** (boto3.client.Client) – The client to get the region from.

Returns AWS region string.

Return type string

stacker.util.**get_config_directory**()
Return the directory the config file is located in.

This enables us to use relative paths in config values.

stacker.util.**get_hosted_zone_by_name**(*client*, *zone_name*)
Get the zone id of an existing zone by name.

Parameters

- **client** (botocore.client.Route53) – The connection used to interact with Route53's API.
- **zone_name** (*string*) – The name of the DNS hosted zone to create.

Returns The Id of the Hosted Zone.

Return type string

stacker.util.**get_or_create_hosted_zone**(*client*, *zone_name*)
Get the Id of an existing zone, or create it.

Parameters

- **client** (botocore.client.Route53) – The connection used to interact with Route53's API.
- **zone_name** (*string*) – The name of the DNS hosted zone to create.

Returns The Id of the Hosted Zone.

Return type string

stacker.util.**get_s3_endpoint**(*client*)
Gets the s3 endpoint for the given boto3.client.Client object.

Parameters **client** (boto3.client.Client) – The client to get the endpoint from.

Returns The AWS endpoint for the client.

Return type string

stacker.util.**get_soa_record**(*client*, *zone_id*, *zone_name*)
Gets the SOA record for zone_name from zone_id.

Parameters

- **client** (botocore.client.Route53) – The connection used to interact with Route53's API.
- **zone_id** (*string*) – The AWS Route53 zone id of the hosted zone to query.
- **zone_name** (*string*) – The name of the DNS hosted zone to create.

Returns

An object representing the parsed SOA record returned from AWS Route53.

Return type *stacker.util.SOARecord*

stacker.util.**load_object_from_string**(*fqcn*)
Converts "." delimited strings to a python object.

Given a "." delimited string representing the full path to an object (function, class, variable) inside a module, return that object. Example:

load_object_from_string("os.path.basename") load_object_from_string("logging.Logger") load_object_from_string("LocalClassName")

stacker.util.**merge_map**(*a*, *b*)
Recursively merge elements of argument b into argument a.

Primarly used for merging two dictionaries together, where dict b takes precedence over dict a. If 2 lists are provided, they are concatenated.

stacker.util.**parse_cloudformation_template**(*template*)
Parse CFN template string.

Leverages the vendored aws-cli yamlhelper to handle JSON or YAML templates.

Parameters **template** (*str*) – The template body.

stacker.util.**parse_zone_id**(*full_zone_id*)
Parses the returned hosted zone id and returns only the ID itself.

stacker.util.**read_value_from_path**(*value*)
Enables translators to read values from files.

The value can be referred to with the *file://* prefix. ie:

conf_key: ${kms file://kms_value.txt}

stacker.util.**s3_bucket_location_constraint**(*region*)
Returns the appropriate LocationConstraint info for a new S3 bucket.

When creating a bucket in a region OTHER than us-east-1, you need to specify a LocationConstraint inside the CreateBucketConfiguration argument. This function helps you determine the right value given a given client.

Parameters **region** (*str*) – The region where the bucket will be created in.

Returns The string to use with the given client for creating a bucket.

Return type string

stacker.util.**stack_template_key_name**(*blueprint*)
Given a blueprint, produce an appropriate key name.

Parameters **blueprint** (*stacker.blueprints.base.Blueprint*) – The blueprint object to create the key from.

Returns Key name resulting from blueprint.

Return type string

stacker.util.**uppercase_first_letter**(*s*)
Return string "s" with first character upper case.

stacker.util.**yaml_to_ordered_dict**(*stream*, *loader=<class 'yaml.loader.SafeLoader'>*)
Provides yaml.load alternative with preserved dictionary order.

Parameters

- **stream** (*string*) – YAML string to load.
- **loader** (yaml.loader) – PyYAML loader class. Defaults to safe load.

Returns Parsed YAML.

Return type OrderedDict

stacker.variables module

class stacker.variables.**LookupTemplate**(*template*)
Bases: string.Template

A custom string template we use to replace lookup values

idpattern = '[_a-z][^\\$\\{\\}]*'

pattern = <_sre.SRE_Pattern object>

class stacker.variables.**Variable**(*name*, *value*)
Bases: future.types.newobject.newobject

Represents a variable passed to a stack.

Parameters

- **name** (*str*) – Name of the variable
- **value** (*any*) – Initial value of the variable from the config (str, list, dict)

dependencies()

Returns Stack names that this variable depends on

Return type Set[str]

resolve(*context*, *provider*)
Recursively resolve any lookups with the Variable.

Parameters

- **context** (*stacker.context.Context*) – Current context for building the stack
- **provider** (stacker.provider.base.BaseProvider) – subclass of the base provider

resolved
Boolean for whether the Variable has been resolved.

Variables only need to be resolved if they contain lookups.

value
Return the current value of the Variable.

class stacker.variables.**VariableValue**
Bases: future.types.newobject.newobject

Abstract Syntax Tree base object to parse the value for a variable

dependencies()

classmethod parse(*input_object*)

resolve(*context*, *provider*)

resolved()

Returns Whether value() will not raise an error

Return type bool

simplified()
Return a simplified version of the Value. This can be used to e.g. concatenate two literals in to one literal, or to flatten nested Concatenations

Returns VariableValue

value()

class stacker.variables.**VariableValueConcatenation**
Bases: *stacker.variables.VariableValue*, list

dependencies()

resolve(*context*, *provider*)

resolved()
Returns: bool: Whether value() will not raise an error

simplified()
Return a simplified version of the Value. This can be used to e.g. concatenate two literals in to one literal, or to flatten nested Concatenations

Returns VariableValue

value()

class stacker.variables.**VariableValueDict**
Bases: *stacker.variables.VariableValue*, dict

dependencies()

classmethod parse(*input_object*)

resolve(*context*, *provider*)

resolved()
Returns: bool: Whether value() will not raise an error

simplified()
Return a simplified version of the Value. This can be used to e.g. concatenate two literals in to one literal, or to flatten nested Concatenations

Returns VariableValue

value()

class stacker.variables.**VariableValueList**
Bases: *stacker.variables.VariableValue*, list

dependencies()

classmethod parse(*input_object*)

resolve(*context*, *provider*)

resolved()
Returns: bool: Whether value() will not raise an error

simplified()
Return a simplified version of the Value. This can be used to e.g. concatenate two literals in to one literal, or to flatten nested Concatenations

Returns VariableValue

value()

class stacker.variables.**VariableValueLiteral**(*value*)
Bases: *stacker.variables.VariableValue*

resolved()
Returns: bool: Whether value() will not raise an error

value()

class stacker.variables.**VariableValueLookup**(*lookup_name*, *lookup_data*, *handler=None*)
Bases: *stacker.variables.VariableValue*

dependencies()

resolve(*context*, *provider*)

resolved()
Returns: bool: Whether value() will not raise an error

simplified()
Return a simplified version of the Value. This can be used to e.g. concatenate two literals in to one literal, or to flatten nested Concatenations

Returns VariableValue

value()

stacker.variables.**resolve_variables**(*variables*, *context*, *provider*)
Given a list of variables, resolve all of them.

Parameters

- **variables** (list of *stacker.variables.Variable*) – list of variables
- **context** (*stacker.context.Context*) – stacker context
- **provider** (stacker.provider.base.BaseProvider) – subclass of the base provider

Module contents

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z