Bespin

Release 0.1

Jan 24, 2018

Contents

Bespin is configured via a YAML file that contains Bespin configuration, environment specific configuration, and stack specific configuration.

CHAPTER 1

Layout

The layout of your directory is configured by default to look something like:

```
root/
  bespin.yml
  <stack>.json
  <stack2>.yaml

  <environment1>/
    <stack>-params.json
    <stack2>-params.yaml

  <environment2>/
    <stack>-params.json
    <stack2>-params.yaml
```

So say you have two stacks, one called `app` and one called `dns`, along with only one environment called `dev`:

```
root/
  bespin.yml
  app.json
  dns.json

  dev/
    app-params.json
    dns-params.json
```

and your bespin.yml would look something like:

```
---

environments:
  dev:
    account_id: 0123456789
    vars:
      variable1: value1
```

```
stacks:
  app:
    <options>

  dns:
    <options>
```

Where `<options>` are the options for that stack.

Note: The location of the stack template file is configured by the `stack_json` or `stack_yaml` option. The location of the params file is configured by the `params_json` or `params_yaml` option. Alternatively parameters can be specified inline (inside bespin.yml) using `params_yaml`.

1.1 Bespin

assume_role = *(optional)* string

An iam role to assume into before doing any amazon requests.

The iam role can also be set via the `ASSUME_ROLE` environment variable.

This behaviour can be disabled by setting the `NO_ASSUME_ROLE` environment variable to any value.

chosen_artifact = *(default="")* string

The value of the –artifact option. This is used to mean several things via the tasks

chosen_stack = *(default="")* string

The stack to pass into the task

chosen_task = *(default="list_tasks")* string

The task to execute

config = file

Holds a file object to the specified Bespin configuration file

configuration = any

The root of the configuration

dry_run = *(default=False)* boolean

Don't run any destructive or modification amazon requests

environment = *(optional)* string

The environment in the configuration to use.

When a stack is created the stack configuration is merged with the configuration for this environment.

extra = *(default="")* string

Holds extra arguments after a – when executed from the command line

extra_imports = [[string, string] , . . .]

Any extra files to import before searching for the chosen task

flat = *(default=False)* boolean

Used by the `Show` task to show the stacks as a flat list. Set by `--flat`

no_assume_role = *(default=False)* boolean

Boolean saying if we should assume role or not

1.2 Stack

alerting_systems = *(optional)* { string : <options> }

Configuration about alerting systems for downtime_options

endpoint = **(required)** string

Endpoint of the system

name = "{_key_name_1}"

The name of this system

type = string_choice

The type of this system

verify_ssl = *(default=True)* boolean

Boolean saying whether to verify ssl

artifact_retention_after_deployment = *(default=False)* boolean

Delete old artifacts after this deployment is done

artifacts = { string : <options> }

Options for building artifacts used by the stack

archive_format = *(default="tar")* string_choice

The archive file format to use on the artifact (tar, zip)

cleanup_prefix = *(optional)* string

The prefix to use when finding artifacts to clean up

commands = [<options> , . . .]

Commands that need to be run to generate content for the artifact

compression_type = string_choice

The compression to use on the artifact

files = [<options> , . . .]

Any files to add into the artifact

For example:

```
files:
  - content: "{__stack__.vars.version}"
    path: /artifacts/app/VERSION.txt
```

history_length = integer

The number of artifacts to keep in s3

Note: These only get purged if the stack has `artifact_retention_after_deployment` set to true or if the `clean_old_artifacts` task is run

---

not_created_here = *(default=False)* boolean

Boolean saying if this artifact is created elsewhere

paths = [[string, string] , . . .]

Paths to copy from disk into the artifact

upload_to = string

S3 path to upload the artifact to

auto_scaling_group_name = *(optional)* string

The name of the auto scaling group used in the stack

bespin = any

The Bespin object

build_after = [string, . . .]

A list of stacks that should be built after this one is buildt

build_env = [[string, (string_or_int_as_string)] , . . .]

A list of environment variables that are necessary when building artifacts

build_first = [string, . . .]

A list of stacks that should be built before this one is built

build_timeout = *(default=1200)* integer

A timeout for waiting for a build to happen

command = *(optional)* string

Used by the `command_on_instances` task as the command to run on the instances

confirm_deployment = *(optional)* <options>

Options for confirming a deployment

auto_scaling_group_name = *(optional)* string

The name of the auto scaling group that has the instances to be checked

deploys_s3_path = *(optional)* [[string, (integer)] , . . .]

A list of s3 paths that we expect to be created as part of the deployment

sns_confirmation = *(optional)* <options>

Check an sqs queue for messages our Running instances produced

deployment_queue = **(required)** string

The sqs queue to check for messages

timeout = *(default=300)* integer

Stop waiting after this amount of time

version_message = **(required)** string

The expected version that indicates successful deployment

url_checker = *(optional)* <options>

Check an endpoint on our instances for a particular version message

check_url = **(required)** string

The path of the url to hit

endpoint = **(required)** delayed

The domain of the url to hit

expect = **(required)** string

The value we expect for a successful deployment

timeout_after = *(default=600)* integer

Stop waiting after this many seconds

zero_instances_is_ok = *(default=False)* boolean

Don't do deployment confirmation if the scaling group has no instances

dns = *(optional)* dns

Dns options

downtimer_options = *(optional)* { valid_string(valid_alerting_system) : <options> }

Downtime options for the downtime and undowntime tasks

hosts = [string, ...]

A list of globs of hosts to downtime

env = [[string, (string_or_int_as_string)] , ...]

A list of environment variables that are necessary for this deployment

environment = "{environment}"

The name of the environment to deploy to

ignore_deps = *(default=False)* boolean

Don't build any dependency stacks

key_name = "{_key_name_1}"

The original key of this stack in the configuration['stacks']

name = *(default="{_key_name_1}")* string

The name of this stack

netscaler = *(optional)* <options>

Netscaler declaration

configuration = *(optional)* { string : { string : netscaler_config } }

Configuration to put into the netscaler

configuration_password = *(optional)* string

The password for configuration syncing

configuration_username = *(optional)* string

The username for configuration syncing

dry_run = to_boolean

Whether this is a dry run or not

host = **(required)** string

The address of the netscaler

nitro_api_version = *(default="v1")* string

Defaults to v1

password = delayed

The password

syncable_environments = *(optional)* [valid_environment, ...]

List of environments that may be synced

username = **(required)** string

The username

verify_ssl = *(default=True)* boolean

Whether to verify ssl connections

newrelic = *(optional)* <options>

Newrelic declaration

account_id = **(required)** string

The account id

api_key = **(required)** string

The api key to newrelic

application_id = **(required)** string

The application id

deployed_version = **(required)** string

Deployed version

env = [[string, (string_or_int_as_string)] , ...]

Required environment variables

notify_stackdriver = *(default=False)* boolean

Whether to notify stackdriver about deploying the cloudformation

params_json = valid_params_json

The path to a json file for the parameters used by the cloudformation stack

params_yaml = valid_params_yaml

Either a dictionary of parameters to use in the stack, or path to a yaml file with the dictionary of parameters

role_name = string

The IAM role that cloudformation assumes to create the stack

scaling_options = <options>

Options for the scale_instances command

highest_min = *(default=2)* integer

No description

instance_count_limit = *(default=10)* integer

No description

sensitive_params = *(default=['Password'])* [string, ...]

Used to hide sensitive values during build

skip_update_if_equivalent = [[delayed, delayed] , ...]

A list of two variable definitions. If they resolve to the same value, then don't deploy

ssh = *(optional)* <options>

Options for ssh'ing into instances

address = *(optional)* string

The address to use to get into the single instance if `instance` is specified

auto_scaling_group_name = *(optional)* string

The logical id of the auto scaling group that has the instances of interest

bastion = *(optional)* string

The bastion jumpbox to use to get to the instances

bastion_key_location = *(optional)* string

The place where the bastion key may be downloaded from

bastion_key_path = *(default="{config_root}/{environment}/bastion_ssh_key.pem")* string

The location on disk of the bastion ssh key

bastion_user = **(required)** string

The user to ssh into the bastion as

instance = *(optional)* [string, ...]

The Logical id of the instance in the template to ssh into

instance_key_location = *(optional)* string

The place where the instance key may be downloaded from

instance_key_path = *(default="{config_root}/{environment}/ssh_key.pem")* string

The location on disk of the instance ssh key

storage_host = *(optional)* string

The host for the storage of the ssh key

storage_type = *(default="url")* string_choice

The storage type for the ssh keys

user = **(required)** string

The user to ssh into the instances as

stack_json = valid_stack_json

The path to a json file for the cloudformation stack definition

stack_name = *(default="{_key_name_1}")* string

The name given to the deployed cloudformation stack

Note that this may include environment variables as defined by the `stack_name_env` option:

```
stack_name: "rerun-{{RELEASE_VERSION}}"
stack_name_env:
    - RELEASE_VERSION
```

stack_name_env = [[string, (string_or_int_as_string)] , . . .]

A list of environment variables that are necessary for creating the stack name

stack_policy = valid_policy_json

The path to a json file for the cloudformation stack policy

stack_yaml = valid_stack_yaml

The path to a yaml file for the cloudformation stack definition

stackdriver = *(optional)* <options>

Stackdriver options used for giving events to stackdriver

api_key = **(required)** string

The api key used to gain access to stackdriver

deployment_version = *(default="<version>")* string

The version being deployed

suspend_actions = *(default=False)* boolean

Suspend Scheduled Actions for the stack before deploying, and resume Scheduled actions after finished deploying.

This uses the `auto_scaling_group_name` attribute to determine what autoscaling group to suspend and resume

tags = { valid_string(regex(^.{0,127}$)) : string }

A dictionary specifying the tags to apply to the stack

Cloudformation will apply these tags to all created resources

termination_protection = *(default=False)* boolean

Whether to enable termination protection for the stack

vars = delayed

A dictionary of variable definitions that may be referred to in other parts of the configuration

1.3 Environment

account_id = **(required)** (valid_string(regex(\d+)) or integer)

AWS account id for this environment

region = *(default="ap-southeast-2")* string

AWS region name for this environment

tags = { valid_string(regex(^.{0,127}$)) : string }

A dictionary specifying the tags to apply to the stack

Cloudformation will apply these tags to all created resources

vars = dictionary

A dictionary of variable definitions that may be referred to in other parts of the configuration

1.4 Password

bespin = "{bespin}"

The bespin object

crypto_text = **(required)** string

The encrypted version of the password

encryption_context = *(optional)* dictionary

Any encryption context

grant_tokens = *(optional)* [string, ...]

List of any grant tokens

KMSMasterKey = **(required)** string

The kms master key id

name = "{_key_name_1}"

The name of the password

vars = dictionary

Extra variables

CHAPTER 2

Formatter

Configuration values may reference other parts of the config using 'replacement fields' surrounded by curly braces `{}`. Nested values can be referenced using dot notation, eg: `{foo.bar.quax}`. If you need to include a brace character in the literal text, it can be escaped by doubling: `{{` and `}}`.

Available fields:

environment Current environment name as a string

region Current environment's region

environments.<env_name>.* Environment mappings.

Environment fields includes:

account_id Environment AWS account id

region Environment AWS region

stacks.<stack_name>.* Stack mappings. See *Stack* spec for more detail.

tags.* Tags mapping

vars.* Vars mapping

Within a stack, bespin also defines the following aliases:

__stack_name__ Current stack name as a string.

__stack__ Current stack mapping (ie: stacks.__stack_name__). See *Stack* spec for more detail.

__environment__ Current environment mapping (ie: environments.environment).

In addition to configuration fields, bespin defines the following special values:

config_root Directory of the main configuration file (ie: `dirname` of `--bespin-config`)

:config_dir *(advanced) (python2.7+ or python3 required)*

Directory of the configuration file where the value was defined. See `bespin.extra_files`.

_key_name_X *(advanced)*

Refers to the key's content X positions up from the current value, indexed from zero. For example, the following would result in "example vars test":

```
stacks:
  test:
    vars:
      example: "{_key_name_0} {_key_name_1} {_key_name_2}"
```

Fields may also declare a formatter by suffixing the field with a colon `:` and the name of the formatter to use. Available formatters include:

:env Formats environment variables suitable to be used in shell. `{USER:env}` would produce `${USER}`.

:date Return a string representing the current datetime (`datetime.datetime.now()`) formatted by strftime. See Python strftime for available format codes. eg: `{%Y:date}` would result in the current year (eg: "2017")

:underscored Converts '-' to '_'.

:count Returns the total number of elements in a list or CommaDelimitedList variable as a string.

The total number of elements in a CommaDelimitedList should be one more than the total number of commas. This implementation marries Cloudformation Parameters CommaDelimitedList's implementation. Examples:

```
vars:
  one: "1"          # {one:count} == "1"
  two: "1,2"        # {two:count} == "2"
  three: "1,2,3"    # {three:count} == "3"
  empty: ""         # {empty:count} == "1"
  space: " "        # {space:count} == "1"
  comma: ","        # {comma:count} == "2"
```

Note: The formatter does not support nested values (eg: {a.{foo}.c}). See *Stacks* for details on using variable formatting (ie: XXX_MYVAR_XXX) instead.

CHAPTER 3

Tasks

Bespin's mechanism for doing anything are tasks. By default Bespin comes with a number of tasks as describe below:

3.1 Default tasks

show

Show what stacks we have in layered order.

When combined with the `--flat` option, the stacks are shown as a flat list instead of in layers.

tail Tail the deployment of a stack

deploy Deploy a particular stack

become Print export statements for assuming an amazon iam role

params Print out the params

bastion SSH into the bastion

outputs Print out the outputs

execute Exec a command using assumed credentials

downtime Downtime this stack in alerting systems

instances Find and ssh into instances

list_tasks List the available_tasks

undowntime UnDowntime this stack in alerting systems

deploy_plan Deploy a predefined list of stacks in order

sanity_check Sanity check a stack and it's dependencies

num_instances Count the number of running instances.

print_variable Prints out a variable from the stack

scale_instances Change the number of instances in the stack's auto_scaling_group

encrypt_password Convert plain text password into crypto text

sanity_check_plan sanity check a predefined list of stacks in order

publish_artifacts Build and publish an artifact

confirm_deployment Confirm deployment via SNS notification for each instance and/or url checks

validate_templates Validates all stack templates and parameters against CloudFormation

wait_for_dns_switch Periodically check dns until all our sites point to where they should be pointing to for specified environment

clean_old_artifacts Cleanup old artifacts

command_on_instances Run a shell command on all the instances in the stack

sync_netscaler_config Sync netscaler configuration with the specified netscaler

switch_dns_traffic_to Switch dns traffic to some environment

create_stackdriver_event Create an event in stackdriver

enable_server_in_netscaler Disable a server in the netscaler

disable_server_in_netscaler Enable a server in the netscaler

note_deployment_in_newrelic Note the deployment in newrelic

resume_cloudformation_actions Resumes all schedule actions on a cloudformation stack

suspend_cloudformation_actions Suspends all schedule actions on a cloudformation stack

3.2 Custom Tasks

There are two ways you can create custom tasks.

The first way is to define `tasks` as part of a stack definition:

```
---

stacks:
  app:
    [..]

    tasks:
      deploy_app:
        action: deploy
```

Will mean that you can run `bespin deploy_app dev` and it will run the `deploy` action for your `app` stack.

Tasks have several options:

action The task to run. Note that the stack will default to the stack you've defined this task on.

options Extra options to merge into the stack configuration when running the task.

overrides Extra options to merge into the root of the configuration when running the task.

description A description that is shown for this task when you ask Bespin to list all the tasks.

The second way of defining custom tasks is with the `extra_imports` option.

For example, let's say you have the following layout:

```
bespin.yml
app.json
scripts.py
```

And your bespin.yml looked like:

```
---

bespin:
  extra_imports:
  - ["{config_root}", "scripts"]

stacks:
  app:
    [..]
```

Then before Bespin looks for tasks it will first import the python module named `scripts` that lives in the folder where `bespin.yml` is defined. So in this case, the `scripts.py`.

The only thing `scripts.py` needs is a `__bespin__(bespin, task_maker)` method where `bespin` is the `Bespin` object and `task_maker` is a function that may be used to register tasks.

For example:

```
def __bespin__(bespin, task_maker):
  task_maker("deploy_app", "Deploy the app stack", action="deploy").specify_
↪stack("app")
```

Here we have defined the `deploy_app` action that will `deploy` the `app` stack.

We can do something more interesting if we also define a custom action:

```
from bespin.tasks import a_task

def __bespin__(bespin, task_maker):
  task_maker("list_amis", "List amis with a particular tag")

@a_task(needs_credentials=True)
def list_amis(overview, configuration, **kwargs):
  credentials = configuration['bespin'].credentials
  amis = credentials.ec2.get_all_images(filters={"tag:application": "MyCreatedAmis"})
  for ami in amis:
    print(ami.id)
```

And then we can do `bespin list_amis dev` and it will find all the Amis that have an `application` tag with `MyCreatedAmis`.

CHAPTER 4

Stacks

Bespin revolves around the concept of a cloudformation stack. Defining them is one of the required options in the *Configuration*.

A cloudformation stack has two parts to it:

The template file Cloudformation is defined by a template file - see Cloudformation template basics

Currently bespin supports the JSON and YAML Cloudformation formats.

The parameters Cloudformation has the idea of parameters, where you define variables in your stack and then provide values for those variables at creation time.

Bespin provides the option of either specifying a file containing these values or, more conveniently, you may specify them inline with the configuration as a yaml dictionary.

So if you have the following directory structure:

```
/my-project/
  bespin.yml
  app.json
  params.json
```

And the following configuration:

```
---

environments:
  dev:
    account_id: "123456789"

stacks:
  app:
    stack_name: my-application
    stack_json: "{config_root}/app.json"
    params_json: "{config_root}/params.json"
```

Then `bespin deploy dev app` will deploy the `app.json` using `params.json` as the parameters.

Where `params.json` looks like:

```
[ { "ParameterKey": "Key1"
  , "ParameterValue": "Value1"
  }
, { "ParameterKey": "Key2"
  , "ParameterValue": "Value2"
  }
]
```

An equivalent `params.yaml` file would look like:

```
---

Key1: Value1
Key2: Value2
```

Alternatively you can have inline the parameters like so:

```
---

environments:
  dev:
    account_id: "123456789"

stacks
  app:
    stack_name: my-application
    stack_json: "{config_root}/app.json"

    params_yaml:
      Key1: Value1
      Key2: Value2
```

Note: The stack_json and stack_yaml will default to "{config_root}/{_key_name_1}.json" and "{config_root}/{_key_name_1}.yaml". This means if your stack json is the same name as the stack and next to your configuration, then you don't need to specify `stack_json`.

4.1 Defining variables

You can refer to variables defined in your configuration inside params_yaml using a `XXX_<VARIABLE>_XXX` syntax. So if you have defined a variable called `my_ami` then `XXX_MY_AMI_XXX` inside your params_yaml values will be replaced with the value of that variable.

Note: This syntax is available in addition to the *Configuration Formatter*. Formatter `{}` syntax will only reference config values, and gets interpreted when loading the configuration. Whereas the `XXX_<VARIABLE>_XXX` variable may be sourced from elsewhere (see below: *dynamic variables*, *environment variables*) and can be replaced at runtime.

So let's say I have the following configuration:

```
---
```

```
vars:
  azs: "ap-southeast-2a,ap-southeast-2b"

environments:
  dev:
    account_id: "123456789"
    vars:
      vpcid: vpc-123456

  prod:
    account_id: "987654321"
    vars:
      vpcid: vpc-654321

stacks:
  app:
    stack_name: my-application
    vars:
      ami: ami-4321

    environments:
      dev:
        vars:
          min_size: 0

      prod:
        vars:
          min_size: 2

    params_yaml:
      ami: XXX_AMI_XXX
      AZs: XXX_AZS_XXX
      VpcId: XXX_VPCID_XXX
      MinSize: XXX_MIN_SIZE_XXX
```

Then you'll get the following outputs:

```
$ bespin params dev app
my-application
[
    {
        "ParameterValue": "vpc-123456",
        "ParameterKey": "VPCId"
    },
    {
        "ParameterValue": "ap-southeast-2a,ap-southeast-2b",
        "ParameterKey": "AZs"
    },
    {
        "ParameterValue": "ami-4321",
        "ParameterKey": "ami"
    }
]

$ bespin params prod app
my-application
[
    {
```

```
        "ParameterValue": "vpc-654321",
        "ParameterKey": "VPCId"
    },
    {
        "ParameterValue": "ap-southeast-2a,ap-southeast-2b",
        "ParameterKey": "AZs"
    },
    {
        "ParameterValue": "ami-4321",
        "ParameterKey": "ami"
    }
]
```

If you're looking closely enough you may notice that there is a hierarchy of variables in the configuration. Bespin will essentially collapse this hierarchy into one dictionary of variables at runtime before using them.

The order is:

```
<root>
<environment>
<stack>
<stack_environment>
```

Where values of the same name are overridden.

This allows you to have:

- Variables across all stacks for all environments
- Variables across all stacks for particular environments
- Variables specific to a stack for all environments
- Variables specific to a stack for particular environments

Note: The XXX_<VARIABLE>_XXX syntax is a search and replace, so you can do something like:

```
---

environments:
  dev:
    account_id: "123456789"
    vars:
      subnet_a: subnet-12345
      subnet_b: subnet-67890

stacks:
  app:
    stack_name: my-application

    params_yaml:
      subnets: XXX_SUBNET_A_XXX,XXX_SUBNET_B_XXX
```

and reference more than one variable and intermingle with other characters.

4.2 Dynamic Variables

When you define a variable, you may also specify a list of two items:

```
---

vars:
  vpcid: [vpc-base, VpcId]
  zoneid: ["{stacks.dns-public}", ZoneId]
```

This is a special syntax and stands for `[<stack_name>, <output_name>]` and will dynamically find the specified Cloudformation output for that stack.

If the stack is in bespin's config it can be referenced directly using the *Configuration Formatter*, ie: `["{stacks.my_stack}", <output_name>]`. This will use the `stack_name` from `my_stack` and also add `my_stack` to this stack's `build_first` dependencies.

For those unfamiliar with cloudformation, it allows you to define Outputs for your stacks. These outputs are essentially a Key-Value store of template defined strings.

So in the example above, the `vpcid` variable would resolve to the `VpcId` Output from the `vpc-base` cloudformation stack in the environment being deployed to.

4.3 Environment Variables

You may populate variables with environment variables.

First you must specify `env` as a list of environment variables that need to be defined and then you may refer to them using `XXX_<VARIABLE>_XXX`.

For example:

```
---

environments:
  dev:
    account_id: "123456789"

stacks:
  app:
    stack_name: my-application

    env:
      - BUILD_NUMBER
      - GIT_COMMIT

    params_yaml:
      Version: app-XXX_BUILD_NUMBER_XXX
```

Environment variables can also be defined with defaults or overrides.

"BUILD_NUMBER" No default is specified, so if this variable isn't in the environment at runtime then bespin will complain and quit.

"BUILD_NUMBER:123" A default has been specified, so if it's not in the environment at runtime, bespin will populate this variable with the value "123"

"BUILD_NUMBER=123" An override has been specified. This means that regardless of whether this environment variable has been specified or not, it will be populated with the value of "123"

Note: To use environment variables in `stack_name` refer to Stack's `stack_name` and `stack_name_env` *Configuration* documentation.

4.4 Passwords

Bespin configuration can store KMS encrypted passwords. Environments can have different passwords, and optionally a different encryption key. If an environment `KMSMasterKey` override is provided a new `crypto_text` must obviously also be provided.

Example config:

```
---

environments:
  dev:
    account_id: 123456789
  prod:
    account_id: 987654321

passwords:
  my_secure_password:
    KMSMasterKey: "arn:aws:kms:ap-southeast-2:111111111:alias/developer_key"
    crypto_text: "EXAMPLEZdnUptmwQqlCnQIBEIAewbM7Amw786ZMGBzvqtpnWmK/Ou0jc3RygppQypuB"

    # environment 'prod' override
    prod:
      KMSMasterKey:␣
↪"arn:aws:kms:ap-southeast-2:111111111:key/f65a25e4-1234-4195-8398-a4fcd2ba9c3f"
      crypto_text: "EXAMPLExCDgTs6i+kaQIBEIAef3P/39KEDRafROn0x+PkKZDH9JLPPBnTaVXz+KPj"
```

Passwords can be referenced via `{passwords.name.crypto_text}` and the correct value for the environment will be used.

Passwords can be encrypted using `bespin encrypt_password [environment] [name]`. The user will be prompted to enter the plaintext password via Python getpass and then bespin will encrypt using the `passwords.name` configuration for `environment` and output the `crypto_text` to stdout.

4.4.1 Password decryption

Warning: Care should be taken when passing around decrypted passwords as bespin makes **no effort** to ensure the password is not logged.

Bespin has support for decrypting passwords, though extreme caution should be taken when doing so. Under best practice, **decrypted passwords should NOT be referenced in bespin configuration**.

Cloudformation parameters should always be passed in their encrypted form and decrypted inside Cloudformation using Custom Resources (if needed).

Users implementing *custom task* code can reference the plaintext decryption via `passwords.name.decrypted`.

CHAPTER 5

Deployment

Bespin offers the ability to deploy stacks, taking into account dependency resolution and deployment checking.

For example, let's say we have the following configuration:

```
---

environment:
  dev:
    account_id: "12345789"

stacks:
  security_groups:
    stack_name: appplication_security_groups

  app:
    stack_name: application

    vars:
      app_security_groups: ["{stacks.security_groups}", "AppSecurityGroup"]

    params_yaml:
      AppSecurityGroup: XXX_APP_SECURITY_GROUP_XXX

    build_after:
      - dns

  dns:
    stack_name: application-dns
```

And we do `bespin deploy dev app`, then it will first deploy `security_groups`, use the output from that stack as a variable for the parameters for the `app` stack, which gets deployed next. After the app stack is deployed, the `dns` stack will then be deployed (because of the `build_after` option).

5.1 Plans

You can explicitly specify an order of stacks by creating a `plan`:

```
---

environments:
  dev:
    account_id: "12345678"

plan:
  all:
    - vpc
    - gateways
    - subnets
    - subnet_rules
    - nat
    - dns
    - dhcp
    - dns_names
    - peering

stacks:
  vpc:
    [..]

  gateways:
    [..]

  [..etc..]
```

And then you may deploy that plan with `bespin deploy_plan dev all`

5.2 Confirming deployment

It's useful to be able to confirm that a deployment was actually successful even if the cloudformation successfully deployed:

```
---

environments:
  dev:
    account_id: "123456789"

stacks:
  app:
    stack_name: application

    env:
      - BUILD_NUMBER

    params_yaml:
      BuildNumber: XXX_BUILD_NUMBER_XXX

    confirm_deployment:
```

```
    url_checker:
      expect: "{{BUILD_NUMBER}}"
      endpoint: ["{stacks.app}", PublicEndpoint]
      check_url: /diagnostic/version
      timeout_after: 600
```

In this example, the deployment is checked by checking that a url returns some expected value. In this case it expects the url `/diagnostic/version` to return the BUILD_NUMBER we deployed with.

Confirm_deployment has multiple options

url_checker As per the example above, this checks a url on our app returns a particular value

sns_confirmation: This confirms that an sqs topic receives a particular message:

```
confirm_deployment:
  auto_scaling_group_name: AppServerAutoScalingGroup

  sns_confirmation:
    timeout: 300
    version_message: "{{BUILD_NUMBER}}"
    deployment_queue: deployment-queue
```

This configuration will expect that the sqs queue called `deployment-queue` will receive a message for each new instance in the auto scaling group saying `<instance_id>:success:<version_message>`

Actually sending these messages is up to the definition of the cloudformation stack.

Note: The naming of this is the result of an implementation detail where this was first implemented for a stack that populated the sqs queue via an sns notification.

---

deploys_s3_path: This allows you to specify an s3 path that you expect to have a value with a modified time newer than the deployment of the stack:

```
confirm_deployment:
  deploys_s3_path:
    - ["s3://my-bucket/generated/thing.tar.gz", 600]
```

Where the number is the timeout of looking for this s3 path.

5.3 When zero instances is ok

In some environments it may be ok that a stack deploys and has no instances associated with it. In this case you may set the `zero_instances_is_ok: true`.

If this isn't set and no instances are in the autoscaling group after the stack is deployed, then Bespin will complain saying the deployment failed to make any instances:

```
---

environments:
  dev:
    account_id: "123456789"

  prod:
```

```
    account_id: "123456789"

stacks:
  app:
    stack_name: my-application

    confirm_deployment:
      auto_scaling_group_name: AppServerAutoScalingGroup

      url_checker:
        endpoint: endpoint.my-company.com
        expects: success
        check_url: /diagnostic/status

    # Add zero_instances_is_ok just for the dev environment
    environments:
      dev:
        confirm_deployment:
          zero_instances_is_ok: true
```

CHAPTER 6

Artifacts

Bespin lets you define, create and upload artifacts as defined in the configuration. Where an artifact is just an archive of files either generated or taken from the filesystem.

Artifacts are defined per *stack*:

```
---

environments:
  dev:
    account_id: "123456789"

stacks:
  app:
    artifacts:
      main:
        compression_type: gz

        upload_to: s3://my-bucket/artifacts/main.tar.gz

        paths:
          - ["{config_root}/ansible", "/ansible"]
```

With this example, `bespin publish_artifacts dev app` will create an archive of an `ansible` folder next to the configuration, which is uploaded to `s3://my-bucket/artifacts/main.tar.gz`.

6.1 Specifying the contents

There are currently a few ways of specifying the contents of the archive:

paths As in the example above, paths is a list of lists. Each item in the list being `[<local_location>, <location_in_archive>]` and will take from the local location and put into the archive under the location that is specified.

files Allows you to add files into the archive. For example:

```
files:
  - content: |
      A file
      with content
      goes here
    dest: /location/in/archive.txt
```

This creates a file at `/location/in/archive.txt` with the content as specified.

You can also generate the content from a custom *task*. So say you've defined a custom task called `generate_ansible_playbook` then you can specify:

```
files:
  - task: generate_ansible_playbook
    dest: /ansible/playbook.yml
```

commands This one lets you copy files from your disk into some temporary location, edit any files as you see fit, run an arbitrary command in the temporary location and add files from there into the archive:

```
commands:
  - copy:
      - ["{config_root}/../../play-app", "/"]
    modify:
      "conf/application.conf":
        append:
          - 'app_version="{__stack__.vars.version}"'
    command: "sbt dist"
    add_into_tar:
      - ["target/universal/{vars.app_name}-SNAPSHOT.zip",␣
↪"/artifacts/{vars.app_name}.zip"]
```

Here we've copied our play-app into the root of the temporary location, added the version to the `application.conf`, run `sbt dist` in the temporary location, and then added the resulting file into the archive under `/artifacts/<app_name>.zip`

6.2 Environment Variables

It's useful to be able to pass in environment variables, like the build number and then use it. This is done with `build_env`, which acts like *env*

For example:

```
---

environments:
  dev:
    account_id: "123456789"

stacks:
  app:
    build_env:
      - BUILD_NUMBER
      - GIT_COMMIT

    vars:
```

```
      version: "{{BUILD_NUMBER}}-{{GIT_COMMIT}}"

    artifacts:
      main:
        upload_to: "s3://my-bucket/artifacts/app-{{BUILD_NUMBER}}.tar.gz"

        files:
          - content: {__stack__.vars.version}
            dest: /artifacts/version.txt

        paths:
          - ["{config_root}/ansible", /ansible]
```

Note that referring to environment variables is done with "{{<variable>}}". This is because bespin formats the string twice, once with the configuration, and a second time with the environment variables.

6.3 Cleaning up artifacts

It's dangerous to clean up artifacts with a time based policy in S3 because if you don't create new artifacts for a long enough amount of time, then s3 will clean up an artifact that is used by production and so when new machines come up there won't be an artifact.

Instead, it is better to manually clean up artifacts and keep a certain number of previous artifacts.

Bespin helps this with the `clean_old_artifacts` task:

```
---

environments:
  dev:
    account_id: "123456789"

stacks:
  app:
    build_env:
      - BUILD_NUMBER

    artifacts:
      main:
        history_length: 5
        cleanup_prefix: app-

        compression_type: gz
        upload_to: "s3://my-bucket/artifacts/app-{{BUILD_NUMBER}}.tar.gz"

        paths:
          - ["{config_root}/ansible", /ansible]
```

With this configuration, `bespin clean_old_artifacts dev app` will find all the artifacts under `s3://my-bucket/artifacts` with the prefix `app-`, keep the newest `5` and delete the rest.

Note: If you just want to use the clean_old_artifacts logic but your artifacts are generated and uploaded by something else, then specify `not_created_here: true`

CHAPTER 7

SSH'ing into instances

It's useful to be able to ssh into instances that your bring up in your stack.

Note: bespin uses RadSSH which honours `ssh_config(5)` (ie: ~/.ssh/config). Users may want to set `StrictHostKeyChecking no` to ignore hostkeys and/or `UserKnownHostsFile /dev/null` to prevent host key additions for dynamic/cloud instances.

Bespin provides the `instances` command for finding the instances, getting the ssh key, and ssh'ing into one of the instances.

This command also handles going via a jumphost/bastion instance.

```
---

environments:
  dev:
    account_id: "123456789"

stacks:
  app:
    stack_name: my_application

    ssh:
      bastion_host: bastion.my_company.com
      bastion_user: ec2-user
      bastion_key_path: "{config_root}/{environment}/bastion_ssh_key.pem"

      user: ec2-user
      auto_scaling_group_name: AppServerAutoScalingGroup
      instance_key_path: "{config_root}/{environment}/ssh_key.pem
```

With this configuration, `bespin instances dev app` will look for all the instances in the `AppServerAutoScalingGroup` defined by the `my_application` cloudformation stack and list the ips:

```
$ bespin instances dev app
Found 1 instances
====================
i-d848ca04      10.35.3.151     running Up 9990 seconds
```

Then you can run `bespin instances dev app 10.35.3.151` and with this configuration will ssh through `ec2-user@bastion.my_company.com` into `ec2-user@10.35.3.151`.

If the bastion options are not specified, then no bastion is used.

7.1 Fetching ssh keys from Rattic

Bespin offers the ability to fetch ssh keys stored in Rattic:

```
---

environments:
  dev:
    account_id: "123456789"

stacks:
  app:
    stack_name: my_application

    ssh:
      bastion_host: bastion.my_company.com
      bastion_user: ec2-user
      bastion_key_path: "{config_root}/{environment}/bastion_ssh_key.pem"
      bastion_key_location: "2200"

      user:ec2-user
      auto_scaling_group_name: Appserverautoscalinggroup
      instance_key_location: "2201"

      storage_type: rattic
      storage_host: rattic.my_company.com
      instance_key_path: "{config_root}/{environment}/ssh_key.pem
```

With this configuration, if bespin can't find the ssh key specified by `bastion_key_path` and `instance_key_path` then it will get the ssh keys from `rattic.my_company.com` using the key ids specified by `bastion_key_location` and `instance_key_location`.

Note that the ssh keys must be uploaded to rattic as ssh keys, not as attachments.

Note: The instance_key_path and bastion_key_path in these two examples are the same as the defaults, so leaving them out would have the same effect.

7.2 Specifying hosts

The hosts can be found by either specifying `auto_scaling_group_name` which will look for all the instances attached to that scaling group, or by specifying `instance` which will look for that instance as specified in the cloudformation stack.

For example, if my stack.json has this in it:

```
{ "Resources":
  { "MyInstance":
    { "Type": "AWS::EC2::Instance"
    , "Properties": [..]
    }
  }
}
```

Then I can specify it by having:

```
ssh:
  user: ec2-user
  instance: MyInstance
```

When you do this you may also specify an address that is displayed instead of an ip address:

```
ssh:
  user: ec2-user
  instance: BastionHost
  address: bastion.{environment}.my-company.com
```

So you'd get something like:

```
$ bespin instances dev app
Found 1 instances
====================
i-d848ca04      bastion.dev.my-company.com     running Up 9001 seconds

$ bespin instances prod app
Found 1 instances
====================
i-f849ca94      bastion.prod.my-company.com     running Up 9001 seconds
```

CHAPTER 8

Project Dormant

This project was purpose built for a need at a previous workplace of mine.

As far as I could tell, that was the only place this was used and since my departure from that workplace they have slowly switched to other projects for their deployments.

I don't use AWS at my current workplace, or in any personal projects and so haven't needed to use this.

In the future it's likely I'll only have time for small changes in this project.

CHAPTER 9

Bespin

An opinionated wrapper around Amazon Cloudformation that reads yaml files. and make things happen. 

The documentation can be found at http://bespin.readthedocs.io

9.1 Installation

Just use pip:

```
pip install bespin
```

9.2 Usage

Once bespin is installed, there will be a new program called `bespin`.

When you call bespin without any arguments it will print out the tasks you have available.

You may invoke these tasks with the `task` option.

9.3 Simpler Usage

To save typing `--task`, `--stack` and `--environment` too much , the first positional argument is treated as `task` (unless it is prefixed with a `-`); the second positional argument (if also not prefixed with a `-`) is taken as the `environment` and the third is treated as the `stack`.

So:

```
$ bespin --task deploy --environment dev --stack app
```

Is equivalent to:

```
$ bespin deploy dev app
```

9.4 Logging colors

If you find the logging output doesn't look great on your terminal, you can try setting the `term_colors` option in `bespin.yml` to either `light` or `dark`.

9.5 The yaml configuration

Bespin reads everything from a yaml configuration. By default this is a `bespin.yml` file in the current directory, but may be changed with the `--bespin-config` option or `BESPIN_CONFIG` environment variable.

It will also read from `~/.bespin.yml` and will be overridden by anything in the configuration file you've specified.

9.6 Tests

Install testing deps and run the helpful script:

```
pip install -e .
pip install -e ".[tests]"
./test.sh
```