bang Documentation

Release 0.18

John Calixto

April 03, 2015

Contents

The beginning of the universe...

CHAPTER 1

Overview

Bang automates deployment of server-based software projects.

Projects often comprise multiple servers of varying roles and in varying locations (e.g. traditional server room, cloud provider, multi-datacenter), public cloud resources like storage *buckets* and message queues and other IaaS/PaaS/Splat_aaS resources. DevOps teams already use several configuration management tools like Ansible, Salt Stack, Puppet and Chef to automate on-server configuration. There are also cloud resource *orchestration* tools like CloudFormation and Orchestra/Juju that can be used to automate cloud resource provisioning. Bang combines orchestration with on-server configuration management to provide one-shot, automated deployment of entire project *stacks*.

Bang instantiates cloud resources (e.g. AWS EC2/OpenStack Nova server instances), then leverages Ansible for configuration of all servers whether they are in a server room in the office, across the country in a private datacenter, or hosted by a public cloud provider.

Read the latest online documentation or browse through examples of stack configurations and playbooks.

CHAPTER 2

User Guide

2.1 Installing Bang

2.1.1 Overview

Bang is published in PyPI and can be installed via pip:

```
pip install bang
```

However, Bang depends on other libraries for such things as cloud provider integration and configuration management. The OpenStack client libraries in particular have extra dependencies that can be tricky to install (e.g. python-reddwarfclient depends on lxml).

2.1.2 Installing Dependencies

Debian/Ubuntu

... Using System Packages

Warning: This will likely upgrade some of your system Python packages. E.g. On a stock Ubuntu 12.04 LTS installation, it upgrades boto.

The benefit of installing Bang into your system Python installation is that you don't need to build the native extensions in Bang's dependencies - you can just use the prebuilt packages for your system. The following commands will install Bang to your system Python installation:

```
sudo apt-get install python-pip python-lxml
sudo pip install bang
```

... In a Virtualenv

Unfortunately, some of Bang's dependencies have native extensions that require extra headers and compilation tools. Install the build-time dependencies from the Debian/Ubuntu package repos:

```
sudo apt-get install build-essential python-dev libxml2-dev libxslt-dev
```

Then install Bang as directed above.

OSX

... Using System Packages

... In a Virtualenv

Install build-time dependencies with Homebrew, and link them: brew install libxml2 brew install libxslt brew link libxml2 –force brew link libxslt –force

Then install Bang as directed above

RightScale

Bang allows you to combine traditional cloud providers like AWS with higher-level cloud managers like RightScale in the same stack. Generally, RightScale provides ample automation on top of AWS. However, it is sometimes necessary to supplement RightScale's features. E.g.

- Using new AWS technologies that are not yet supported by RightScale.
- Integrating inherited application stacks that do not use Chef.
- Working with resources in multiple public cloud providers or even traditional private datacenters.

To enable the `rightscale` provider, install the following dependency:

```
pip install python-rightscale==0.1.3
```

OpenStack

As much as possible, Bang uses official OpenStack client libraries to provision resources in OpenStack clouds. Prior to Bang 0.10, this dependency was explicitly defined in the Bang package such that `pip install bang` would install the OpenStack client libraries as well. From Bang 0.10 onwards, OpenStack users will need to install the client libraries on their own.

Note: Problems with the client libraries include:

- Not having dependencies defined correctly in their packages
- Unnecessary dependency on native libraries like lxml

Bugs have been filed with upstream, but they have not been very responsive to feedback from outside the OpenStack organization.

The following commands should install the necessary dependencies:

```
sudo apt-get install build-essential python-dev libxml2-dev libxslt-dev
pip install \
    python-novaclient==2.11.1
    python-swiftclient==1.3.0
    python-reddwarfclient==0.1.2
    novaclient-auth-secretkey
```

HP Cloud

HP Cloud uses OpenStack as a base cloud operating system. However, HP has its own proprietary extensions and modifications which have meaningful effects on the provisioning API. Bang subclasses the appropriate OpenStack client library classes and adjusts behaviour for HP Cloud. In addition to the OpenStack dependency installation listed above, the following commands will enable Bang to deploy databases to HP Cloud's beta DBaaS:

```
pip install PyMySQL==0.5
```

2.2 Running Bang

2.2.1 Quick Start

With all of your deployer credentials (e.g. AWS API keys) and stack configuration in the same file, `mywebapp.yml`, you simply run:

```
bang mywebapp.yml
```

As a convenience for successive invocations, you can set the BANG_CONFIGS environment variable:

```
export BANG_CONFIGS=mywebapp.yml

# Deploy!
bang

# ... Hack on mywebapp.yml
```

```
# Deploy again!
bang

# ... Uh-oh, connection issues on one of the hosts.  Could be
# transient interweb goblins - deploy again!
bang

# Yay!
```

2.2.2 BANG_CONFIGS

Set this to a colon-separated list of configuration specs.

2.2.3 Other Options

Deploys a full server stack based on a stack configuration file.

In order to SSH into remote servers, ‘‘bang‘‘ needs the corresponding private key for the public key specified in the ‘‘ssh_key_name‘‘ fields of the config file. This is easily managed with ssh-agent, so ‘‘bang‘‘ does not provide any ssh key management features.

```
usage: bang [-h] [--ask-pass] [--user USER] [--dump-config {json,yaml,yml}]
            [--list] [--no-configure] [--no-deploy] [--playbook PLAYBOOK]
            [--version]
            [CONFIG_SPEC [CONFIG_SPEC ...]]
```

Positional arguments:

config_specs Stack config specs(s).

A *config spec* can either be a basename of a config file (e.g. ‘‘mynewstack‘‘), or a path to a config file (e.g. ‘‘../bang-stacks/mynewstack.yml‘‘).

A basename is resolved into a proper path this way:

- Append ‘‘.yml‘‘ to the given name. - Search the ‘‘config_dir‘‘ path for the resulting filename, where the value for ‘‘config_dir‘‘ comes from ‘‘$HOME/.bangrc‘‘.

When multiple config specs are supplied, the attributes from all of the configs are deep-merged together into a single, *union* config in the order specified in the argument list.

If there are collisions in attribute names between separate config files, the attributes in later files override those in earlier files.

At deploy time, this can be used to provide secrets (e.g. API keys, SSL certs, etc...) that you don't normally want to check in to version control with the main stack configuration.

Options:

--ask-pass=False, -k=False ask for SSH password

--user, -u set SSH username (default=docs)

--dump-config Dump the merged config in the given format, then quit

Possible choices: json, yaml, yml

--list=False Dump stack inventory in ansible-compatible JSON.

Be sure to set the ''BANG_CONFIGS'' environment variable to a colon-separated list of config specs.

E.g.

specify the configs to use export BANG_CONFIGS=/path/to/mystack.yml:/path/to/secrets.yml

dump the inventory to stdout bang –list

run some command ansible webservers -i /path/to/bang -m ping

--no-configure=True Do *not* configure the servers (i.e. do *not* run the ansible playbooks).

This allows the person performing the deployment to perform some manual tweaking between resource deployment and server configuration.

--no-deploy=True Do *not* deploy infrastructure resources.

This allows the person performing the deployment to skip creating infrastructure and go straight to configuring the servers. It should be obvious that configuration may fail if it references infrastructure resources that have not already been created.

--playbook, -p Specify playbook(s) to run during the Ansible phase.

WARNING This overrides any list of playbooks specified in the bang config(s).

This argument can be passed multiple times to specify multiple playbooks to run. They will be executed in the order in which they are passed on the command line.

E.g.

deploy and configure a stack as usual. playbooks # are defined in ‘‘my_own_cloud.yml‘‘: bang own_cloud.yml

run an ad-hoc playbook on the same stack: bang own_cloud.yml -p update_loadbalancers.yml

run multiple ad-hoc playbooks: bang own_cloud -p start_maintenance_window.yml \ -p restart_apache.yml \ -p stop_maintenance_window.yml

--version, -v show program's version number and exit

2.3 Stack Configurations

2.3.1 Examples

Examples of Bang config files are available with the source code:

https://github.com/fr33jc/bang/tree/master/examples

2.3.2 Config File Structure

The configuration file is a YAML document. Like a play in an Ansible playbook, the outermost data structure is a YAML mapping.

Like Python, blocks/sections/stanzas in a Bang config file are visually defined by indentation level. Each top-level section name is a key in the outermost mapping structure.

There are some reserved Top-Level Keys that have special meaning in Bang and there is an implicit, broader grouping of these top-level keys/sections. The broader groups are:

- General Stack Properties
- Configuration Scopes
- Stack Resource Definitions

Any string that is a valid YAML identifier and is **not** a reserved top-level key is available for use as a *custom configuration scope*. It is up to the user to avoid name collisions between keys, especially between reserved keys and custom configuration scope keys.

2.3.3 Top-Level Keys

General Stack Properties

The attributes in this section apply to the entire stack.

The following top-level section names are reserved:

name This is the unique stack name. E.g. `myblog-prod`, `myblog-staging`, `monitoring`, etc...

version The overall stack version. A stack may be made up of many components each with their own release cycle and versioning scheme. This version could be used as the umbrella version for an entire product/project release.

logging Contains configuration values for Bang's logging.

deployer_credentials See `bang.providers.hpcloud.HPCloud.authenticate()`

playbooks A list of playbook filenames to execute.

Stack Resource Definitions

These configuration stanzas describe the building blocks for a project. Examples of stack resources include:

- Cloud resources
  - Virtual servers
  - Load balancers
  - Firewalls and/or security groups
  - Object storage
  - Block storage
  - Message queues
  - Managed databases
- Traditional server room/data center resources
  - Physical or virtual servers
  - Load balancers
  - Firewalls

Users can use Bang to manage stacks that span across traditional and cloud boundaries. For example, a single stack might comprise:

- Legacy database servers in a datacenter
- Web application servers in an OpenStack public cloud
- Message queues and object storage from AWS (i.e. SQS)

Every stack resource key maps to a *dictionary* for that particular resource type, where the keys are resource names. Each value of the dictionary is a key-value map of attributes. Most attributes are specific to the type of resource being deployed.

Every cloud resource definition must contain a `provider` key whose value is the name of a Bang-supported cloud provider.

Server definitions that do not contain a `provider` key are assumed to be *already provisioned*. Instead of a set of cloud server attributes, these definitions merely contain `hostname` values and the appropriate configuration scopes.

The reserved stack resource keys are described below:

queues E.g. SQS

buckets E.g. S3, OpenStack Swift

databases E.g. RDS, OpenStack RedDwarf

server_security_groups E.g. EC2 and OpenStack Nova security groups

servers E.g. EC2, OpenStack Nova, VPS virtual machines.

load_balancers: E.g. ElasticLoadBalancer, HP cloud LBaaS

Configuration Scopes

Configuration scopes typically define high-level attributes and values that you might want to alter between instantiations of a stack. For example, a blog stack might be made up of some frontend load balancers running haproxy 1.4 that distribute requests to an array of web app servers running version 1.1 of your custom application called *my_blog_app*. The production Bang config would have config scopes like this:

```
my_blog_app:
  version: '1.1'

haproxy:
  version: '1.4'
```

You would reuse the same infrastructure configuration and set of Ansible playbooks to stand up a QA or development stack. When you release version 1.2 of *my_blog_app* you just adjust the value in the config scope like this:

```
my_blog_app:
  version: '1.2'

haproxy:
  version: '1.4'
```

In this example, if you then wanted to test out haproxy 1.5, the config scopes would look like this:

```
my_blog_app:
  version: '1.2'

haproxy:
  version: '1.5'
```

Config scopes can be used for more than just component versions. When deciding what attributes to put in config scopes and what attributes to put into your Ansible variables, consider that Bang config scopes are ideal for values that you might vary per environment or per iteration of an environment.

Since the Bang config files and all of the associated playbooks are just text files, they can be managed the same way you manage your code in a revision control system. You can branch, merge, and tag the same way you do with your application code. With the right tags, it's trivial to compare the config scope values that are in production with those that are in your QA or development environments.

Reusable Definition

Any top-level section name that is not specified above as a reserved key in General Stack Properties or in Stack Resource Definitions, is parsed and categorized as a custom configuration scope. For example, a media transcoding web service might have the following config scopes:

```
apache:
  preforks: 4
  modules:
  - rewrite
  - wsgi

my_web_frontend:
  version: '1.2.0'
  log_level: WARN

my_transcoder_app:
  version: '1.1.5'
  log_level: INFO
  src_types:
  - h.264+aac
  - theora+vorbis
```

The key names and the values are arbitrary and defined solely by the user.

When running the on-server configuration phase of a Bang run, Bang uses the `config_scopes` in a server definition to determine what to pass to Ansible as *inventory variables* for a particular host. To refer to a top-level, reusable config scope in a server definition, list its name like this:

```
# Config Scopes
# -------------
apache:
  preforks: 4
  modules:
  - rewrite
  - wsgi

my_web_frontend:
  version: '1.2.0'
  log_level: WARN


# Resource Definitions
# --------------------
servers:
  web_server:
    # other server attributes go here
    config_scopes:
    - apache
    - my_web_frontend
```

When Ansible runs on the `web_server` hosts, the following references to the config scope variables will be evaluated to their associated values:

```
{{apache.preforks}}          <-- evaluates to 4
{{my_web_frontend.version}}  <-- evaluates to 1.2.0
```

Inline Definition

In addition to the top-level definitions, config scopes for a server may be defined inline. This is mainly useful for simple stacks where reusing config scopes might not be needed. For example:

```
webapp:
  port: 8001
  app_dir: /opt/foo/app

reverse_proxy:
  server_name: newapp.company.com

servers:
  blah:
    config_scopes:
    - webapp
    - reverse_proxy
    - this: is
```

```
      a_config_scope: defined
      inline: yo
```

The config scopes above would make the following inventory variables available to Ansible:

```
{
    'webapp': {
        'port': 8001,
        'app_dir': '/opt/foo/app',
        },

    'reverse_proxy': {
        'server_name': 'huismans.kief.io',
        },

    'this': 'is',

    'a_config_scope': 'defined',

    'inline': 'yo',
}
```

Which would let you use any of the following in playbooks and templates:

```
{{webapp.port}}
{{reverse_proxy.server_name}}
{{this}}
{{a_config_scope}}
{{inline}]
```

2.4 Writing Ansible Playbooks

Bang was written with the goal of being able to use Ansible playbooks either with Bang's builtin playbook runner or directly with `ansible-playbook`. As such, any working Ansible playbook will work when referenced in a Bang config.

Refer to Ansible's playbook documentation for details about writing the actual playbooks.

2.4.1 Search Path

Bang looks for any playbooks referenced by a stack configuration file in a `playbooks/` directory that is a peer of the stack configuration file. After it has found a playbook, it defers to Ansible's path resolution logic for all other includes and file references.

When Ansible searches for modules referenced in a playbook, it allows for playbook-specific modules to live in a `library/` directory that is a *peer of the playbook YAML file*. To supplement this custom module location, Bang sets the Ansible module/library path to a `common_modules/` directory that is a *peer of the stack configuration file*. This means that any custom modules that are used in multiple playbooks (i.e. not just for one specific playbook) can be stored along with your stack configurations, playbooks, templates, etc... in the same directory structure.

2.5 Getting Help

2.5.1 Bang

Search through the mailing list archives or subscribe to bangproject-general@lists.sourceforge.net and post a question/comment.

2.5.2 Ansible

For questions related to `ansible`, `ansible-playbook`, playbooks, and modules, see the Ansible project for documentation and several other support resources.

CHAPTER 3

Releases

3.1 Release Summary

3.1.1 0.18 - March 26, 2015

- Add a logo for the project.
- Add `-p` command line argument to specify playbook(s) on command line.
- RightScale
  - Update dependency to python-rightscale==0.1.3.
  - Tag the rightscale server, not just the instance. This insures instances that launch from the same server definition also get tagged.
  - Allow instance type, AZ, secgroups to be optional for RightScale servers.
  - Expose details of RightScale API error response.

3.1.2 0.17.1 - February 6, 2015

- Ensure stats callback fires.
- RightScale: Fix bug in public_dns_names for newly created server.

3.1.3 0.17 - February 5, 2015

- Add hostvars directly to `--list` output.
  - As an optimization to avoid exec-ing the inventory script for every host, Ansible >= 1.3 accepts the hostvars in the initial inventory dump under a `_meta` key.
- Fix up more python2.6 incompatibilities.

  - This includes addressing the warning about BaseException.message deprecation

- Gracefully handle when $HOME is not in environment.
- RightScale
  - Switch to using public_dns_names.

    This means that in the inventory provided to ansible, hosts will be defined by their public DNS name instead of their public IP address. For RightScale hosts in AWS, this gives you names like `ec2-54-123-45-67.compute-1.amazonaws.com` which gets the magic EC2 DNS resolution (i.e. translates to private address within EC2, translates to public address from outside EC2).

3.1.4 0.15 - November 4, 2014

- Expose bang server attributes to playbooks. E.g. in an ansible template, `{{bang_server_attributes.instance_type}}` might resolve to the value `t1.micro`.
- AWS
  - Fix security group handler. Thanks Sol Reynolds!
- RightScale
  - Support all input types. E.g. `key:`, `cred:`, `env:`, etc...

3.1.5 0.14.1 - October 24, 2014

- Fix console logging level configuration.

3.1.6 0.14 - October 24, 2014

- AWS
  - Add support for creating S3 buckets (Thanks to Sol Reynolds).
  - Add support for IAM roles and other provider-specific server attributes.
- RightScale
  - *BREAKING CHANGE*: Inputs are now nested one level deeper in a server config stanza.

    This was done as part of adding support for provider-specific server attributes. Prior to this change, one would specify the server template inputs in a rightscale server config like this:

```
servers:
  my_rs_server:
    # other server attributes omitted for brevity
    provider: rightscale
    inputs:
      DOMAIN: foo.net
      SOME_OTHER_INPUT: blah blah
```

Provider-specific attributes needed to create/launch servers will now be nested one level deeper in an attribute named after the provider. With this new structure, the corresponding configuration for the example above would look like this:

```
servers:
  my_rs_server:
    # other server attributes omitted for brevity
    provider: rightscale
    rightscale:
      inputs:
        DOMAIN: foo.net
        SOME_OTHER_INPUT: blah blah
```

  - Propagate rs deployment and server name to ec2 tags.

- Issues addressed
  - Fix handling of localhost in inventory
  - #11: Return sorted host lists for `bang --list`.

3.1.7 0.13 - October 17, 2014

- Ansible integration
  - Allow setting some ansible options via bang config or ~/.bangrc:
    * Verbosity (especially for ssh debugging):

      ```
      ansible:
        verbosity: 4
      ```

    * Vault:

      ```
      ansible:
        # ask_vault_pass: true
        # vault_pass: "thisshouldfail"
        vault_pass: "bangbang"
      ```

  - Test against ansible 1.7.2

- Add `--no-deploy` arg to only use existing infrastructure.
- Switch to yaml.safe_load.
- Improve compatibility with Python 2.6, including adding 2.6 as a Travis CI target.

3.1.8 0.12 - August 19, 2014

- Update to ansible >= 1.6.3.
  - Allow ansible vars plugins to work.
- Add RightScale provider.
  - Add server creation and launch support.
  - Expose underlying RightScale response for errors.
  - Implement `create_stack()` to create RightScale *deployments*.
- Reuse existing servers *if possible*. Some scenarios allow a server instance to be found and usable as a deployment target (e.g. bang run failed but server instance launched successfully).
- Allow configuration of logging via ~/.bangrc.
- Add backwards support for python 2.6.
- Reorganize and add new examples.

3.1.9 0.11 - January 8, 2014

- HP Cloud provider
  - *BREAKING CHANGE*: Separate HP Cloud v12 and v13 providers. Users of HP Cloud services must now distinguish between the 2 different API versions of their resources.
  - Add new LB nodes before removing old; fixes error caused by HPCS' rule that a LB must have at least one node.
- Allow load balancers to be region-specific.

3.1.10 Older Releases

0.10.1 - July 22, 2013

- Remove install-time dependency on OpenStack client libraries. Users who need OpenStack/HP Cloud support must now install those libraries independently. *Details...*

0.9 - July 16, 2013

- Update dependencies. Now using:
  - Ansible 1.2
  - logutils >= 3.2
- Fix #4: Set value for "Name" tag on EC2 servers
- Fix EC2 server provisioning

0.8 - May 7, 2013

- AWS provider
  - Create and manage EC2 security groups and their rules.

0.7.1 - April 16, 2013

- Fix installation breakage caused by conflicting dependency statements between python-reddwarfclient and python-novaclient. The resolution was to remove the explicit dependency on `prettytable`.

0.7 - April 12, 2013

- *BREAKING CHANGE*: In a stack config file, the top-level resource definition containers were lists. From 0.7 onward, they must be defined as dictionaries. This allows resource definitions to be deep-merged. The `just_run_a_playbook.yml` example was updated to demonstrate the new config format.

  This change extends the reuse of common config stanzas that was previously only available for *general stack properties* and for *configuration scopes* to *resource definitions*. Prior to this change, the main purpose for this deep-merge behaviour was to allow sysadmins to use a known working dev stack config file and specify a *subset config file* to override secrets (e.g. encryption keys) when deploying production stacks. With the deep-merging of resource definitions, deployers can override any part of the config file and break up their stack configurations into multiple reusable *subset config files* as is most convenient for them. For example, one could easily deploy stack clones in multiple public cloud regions using a single base stack config and a subset stack config for each target region overriding `region_name` in the server definitions.

0.6 - April 3, 2013

- HP Cloud provider

  - Add LBaaS support.
- Add "127.0.0.1" to the inventory to enable local plays.
- Add deployer for externally-deployed servers (e.g. physical servers in a traditional server room, unmanaged virtual servers).
- Reuse ssh connections when running playbooks.
- Allow setting ssh username+password as command-line arguments.

0.5 - March 11, 2013

- Expose server `name` to playbooks as `server_class`

0.4 - March 6, 2013

- Update OpenStack client library dependencies
- Add auto-registration of SSH keys for OpenStack

0.3 - February 11, 2013

- Update ansible dependency to 1.0
- Fix bug that caused a crash when running `bang --list` with a server definition in the stack config for which there was no matching running instance.

0.2 - January 30, 3013

- AWS provider
  - Compute (EC2)
- Inline configuration scopes for server definitions
- Separate regions from availability zones
- Fix multi-region stacks

0.1 - January 15, 2013

- Core Ansible playbook runner
- Parallel cloud resource deployment
- Generic OpenStack provider

- HP Cloud provider
  - Compute (Nova)
    - Including security groups
  - Object Storage (Swift)
  - DBaaS (RedDwarf)

3.2 Road Map

Some of the feature ideas below will be implemented in bang. Some may be better suited for a *bang-utils* project. They're listed here so they won't be forgotten along the way.

3.2.1 General Features

- Allow overriding path to .bangrc via environment variable. This allows external utilities to manage multiple sets of deployer_credentials (e.g. a bangrc per client).
- Add extension/plugin mechanism. At the moment, the mercurial-style (i.e. using an rc-file for registering extensions) is the most palatable because it does *not* demand using setuptools, and because it allows the user to manage files how they please.
  - The corollary is that the setuptools-style (i.e. entry points defined in setup.py) mechanism is *not* desirable.
- In addition to the plugin mechanism, have some hookable events to make integration easier with existing tools that can't easily be converted to plugins. E.g.:

```
exec_hooks:
  pre_deploy:
  post_success:
  - /bin/echo yay
  post_failure:
  - /bin/echo boo
```

- Implement `--dry-run`.
- Validate stack configuration.
  - Check for any build artifacts in the deployment S3 bucket/other central storage location.
- Allow absolute paths to playbooks, or a customizable playbook search path.
- Add playbook parallelization. Allow running multiple playbooks at once. Leave it up to the deployers to sort out inter-playbook dependencies.
- Integrate with revision control system.

  - Autoincrement *stack version* in config file.
  - Tag any config scope that defines a `source_code` attribute.
  - Generate release notes between tags.
- Autoscale servers.
- Add `--destroy` to automate destruction of stacks.
- Support `ansible-playbook` runtime options (e.g. vault and tag values).
- Allow selecting public or private IP addresses for cloud hosts.

3.2.2 Providers

- AWS
  - Add any deployers that don't really apply to less featureful public cloud providers. E.g. SQS, ELB, SNS, etc...
  - Create ssh keypairs if specified by the user in their ~/.bangrc.
  - Add DNS updates via Route53 API.
- Docker/LXC
  - Add Docker and LXC images as base images.
  - Add Docker and LXC containers to Ansible inventories.
  - Use Ansible playbooks to make changes within containers.
- Rackspace
- HP Cloud
  - DB Security Groups
- RightScale
  - Add support for server arrays.

CHAPTER 4

Hacking

4.1 Project Resources

4.1.1 Documentation

The documentation is hosted on *Read the Docs*:

http://bang.readthedocs.org/

4.1.2 Source Code

The stable codeline lives in *GitHub*:

https://github.com/fr33jc/bang

Experimental or otherwise unstable development is also hosted on GitHub in a separate clone:

https://github.com/fr33jc/bang-unstable

Large patches from contributors will be integrated into the unstable repo first. When these new features have been tested and cleaned up appropriately, they will be rebased and promoted to the stable master for release.

4.1.3 Continuous Integration

Pushes to the stable master trigger automated unit testing on *Travis CI*:

https://travis-ci.org/fr33jc/bang

4.2 Design

4.2.1 Configuration File Rationale

The Bang configuration file structure came about with the following goals in mind:

- Readability (by humans)
- Not another bespoke serialization format
- Conciseness

While JSON would allow for there to be one less package dependency, YAML was chosen as the overall serialization format because of its focus on human readability.

In its earliest forms, Bang had its own SSH logic and used Chef for configuration management. When Ansible was identified as being a suitable replacement for the builtin SSH logic and for Chef, it made even more sense to continue using YAML for the file format because users could use the same format for configuring Bang and for authoring Ansible playbooks.

4.2.2 Inventory Persistence

It is often useful to have access to the inventory that was used during a particular Bang run. Bang already provides the inventory and the host variables to Ansible directly as Python objects when executed as `bang`, and as a JSON object output to stdout when executed as an Ansible inventory plugin (i.e. `bang --list`). It should also provide the following features:

- Store inventory and host variables in a user-specified file.
- Create `latest-inventory` symlink to inventory file after each Bang run.
- Allow configuration of inventory output format (i.e. YAML or JSON). Even though the Ansible inventory plugin API uses JSON as the serialization format, Bang's default inventory output should be YAML for symmetry since Bang's input format is YAML as well.
- Allow configuration of above via command-line arguments, ~/.bangrc, or even ansible.cfg.

4.3 API

4.3.1 bang

exception `bang.`**`BangError`**
Bases: `exceptions.Exception`

exception `bang.`**`TimeoutError`**
Bases: `bang.BangError`

4.3.2 bang.attributes

Constants for attribute names of the various resources.

This module contains the top-level config file attributes including those that are typically placed in ~/.bangrc.

bang.attributes.**ANSIBLE = 'ansible'**
A dict containing ansible tuning variables.

bang.attributes.**CONFIG_DIR = 'config_dir'**
The directory in which to look for bang config files using their basenames. E.g. if your config_dir is specified as `$HOME/bang-configs`, the following bang runs are equivalent:

```
bang my_web_app deploy
```

And:

```
bang $HOME/bang-configs/my_web_app.yml $HOME/bang-configs/deploy.yml
```

bang.attributes.**DEPLOYER_CREDS = 'deployer_credentials'**
A dict containing credentials for various cloud providers in which the keys can be any valid provider. E.g. `aws`, `hpcloud`.

bang.attributes.**LOGGING = 'logging'**
The top-level key for logging-related configuration options.

bang.attributes.**NAME = 'name'**
The stack name. Its value is used to tag servers and other cloud resources.

bang.attributes.**NAME_TAG_NAME = 'name_tag_name'**
Like chicken fried chicken... this is a way to configure the name of the tag in which the combined stack-role (a.k.a. *name*) will be stored. By default, unless this is specified directly in ~/.bangrc, the *name* value will be assigned to a tag named "Name" (this is the default tag displayed in the AWS management console). I.e. using Bang defaults, the server named "bar" in the stack named "foo" will have the following tags:

```
stack:  foo
role:   bar
Name:   foo-bar
```

In some cases, admins may have other purposes for the "Name" tag. If ~/.bangrc were to have `name_tag_name` set to `descriptor`, then the server described above would have the following tags:

```
stack:       foo
role:        bar
descriptor:  foo-bar
```

To prevent Bang from assigning the *name* value to a tag, assign an empty string to the `name_tag_name` attribute in ~/.bangrc.

bang.attributes.**PLAYBOOKS = 'playbooks'**
The ordered list of playbooks to run *after* provisioning the cloud resources.

bang.attributes.**PROVIDER = 'provider'**
The resource provider (e.g. `aws`, `hpcloud`). Values for the `provider` attribute will be used to look up the appropriate `Provider` subclass to use when instantiating the associated resource.

bang.attributes.**SERVER_CLASS = 'server_class'**
This is a *derived* attribute that Bang provides for instance tagging, and for Ansible playbooks to consume. It's a combination of the `NAME` and the `VERSION`.

bang.attributes.**STACK = 'stack'**
This is a *derived* attribute that Bang provides for instance tagging, and for Ansible playbooks to consume. It's a combination of the `NAME` and the `VERSION`.

bang.attributes.**VERSION = 'version'**
The stack version. Often, you need a global version of a stack in a playbook. E.g. when a web client wants to query a web service for API compatibility, the playbooks could configure the web service to report this stack version.

4.3.3 bang.attributes.ansible

bang.attributes.ansible.**ASK_VAULT_PASS = 'ask_vault_pass'**
A boolean controlling whether or not to prompt for the vault password

bang.attributes.ansible.**VAULT_PASS = 'vault_pass'**
The string used to decrypt any ansible vaults referenced in playbooks

bang.attributes.ansible.**VERBOSITY = 'verbosity'**
An integer indicating verbosity.

4.3.4 bang.attributes.creds

4.3.5 bang.attributes.database

4.3.6 bang.attributes.loadbalancer

4.3.7 bang.attributes.logging

4.3.8 bang.attributes.secgroup

4.3.9 bang.attributes.server

bang.attributes.server.**BANG_ATTRS = 'bang_server_attributes'**

Provides the server definition from the Bang config as a fact available to the playbooks. E.g. in order to get access to the `disk_image_id` in a playbook:

```
{{bang_server_attributes.disk_image_id}}
```

4.3.10 bang.attributes.ssh_key

4.3.11 bang.attributes.tags

4.3.12 bang.config

class bang.config.**Config**(**args*, ***kwargs*)

Bases: dict

A dict-alike that provides a convenient constructor, stashes the path to the config file as an instance attribute, and performs some validation of the values.

__init__(**args*, ***kwargs*)

Parameters **path_to_yaml** (*str*) – Path to a yaml file to use as the data source for the returned instance.

autoinc()

Conditionally updates the stack version in the file associated with this config.

This handles both official releases (i.e. QA configs), and release candidates. Assumptions about version:

- Official release versions are MAJOR.minor, where MAJOR and minor are both non-negative integers. E.g.

  2.9 2.10 2.11 3.0 3.1 3.2 etc...

- Release candidate versions are MAJOR.minor-rc.N, where MAJOR, minor, and N are all non-negative integers.

  3.5-rc.1 3.5-rc.2

classmethod from_config_specs(*config_specs*, *prepare=True*)

Alternate constructor that merges config attributes from `$HOME/.bangrc` and `config_specs` into a single `Config` object.

The first (and potentially *only* spec) in `config_specs` should be main configuration file for the stack to be deployed. The returned object's `filepath` will be set to the absolute path of the first config file.

If multiple config specs are supplied, their values are merged together in the order specified in `config_specs` - That is, later values override earlier values.

Parameters

- **config_specs** (`list` of `str`) – List of config specs.
- **prepare** (*bool*) – Flag to control whether or not `prepare()` is called automatically before returning the object.

Return type `Config`

prepare()

Reorganizes the data such that the deployment logic can find it all where it expects to be.

The raw configuration file is intended to be as human-friendly as possible partly through the following mechanisms:

- In order to minimize repetition, any attributes that are common to all server configurations can be specified in the `server_common_attributes` stanza even though the stanza itself does not map directly to a deployable resource.
- For reference locality, each security group stanza contains its list of rules even though rules are actually created in a separate stage from the groups themselves.

In order to make the `Config` object more useful to the program logic, this method performs the following transformations:

- Distributes the `server_common_attributes` among all the members of the `servers` stanza.
- Extracts security group rules to a top-level key, and interpolates all source and target values.

validate()

Performs all validation checks on this config.

Raises `ValueError` for invalid configs.

bang.config.**find_component_tarball**(*bucket*, *comp_name*, *comp_config*)
Returns True if the component tarball is found in the bucket.

Otherwise, returns False.

bang.config.**parse_bangrc**()
Parses `$HOME/.bangrc` for global settings and deployer credentials. The `.bangrc` file is expected to be a YAML file whose outermost structure is a key-value map.

Note that even though `.bangrc` is just a YAML file in which a user could store any top-level keys, it is not expected to be used as a holder of default values for stack-specific configuration attributes - if present, they will be ignored.

Returns {} if `$HOME/.bangrc` does not exist.

Return type `dict`

bang.config.**read_raw_bangrc**()

bang.config.**resolve_config_spec**(*config_spec*, *config_dir=''*)
Resolves `config_spec` to a path to a config file.

Parameters

- **config_spec** (*str*) – Valid config specs:
  - The basename of a YAML config file *without* the `.yml` extension. The full path to the config file is resolved by appending `.yml` to the basename, then by searching for the result in the `config_dir`.
  - The path to a YAML config file. The path may be absolute or may be relative to the current working directory. If `config_spec` contains a `/` (forward slash), or if it ends in `.yml`, it is treated as a path.
- **config_dir** (*str*) – The directory in which to search for stack configuration files.

Return type `str`

4.3.13 bang.deployers

Base classes and definitions for bang deployers (deployable components)

bang.deployers.**get_stage_deployers**(*keys*, *stack*)
Returns a list of deployer objects that *create* cloud resources. Each member of the list is responsible for provisioning a single stack resource (e.g. a virtual server, a security group, a bucket, etc...).

Parameters

- **keys** (`Iterable`) – A list of top-level configuration keys for which to create deployers.

- **config** (Stack) – A stack object.

Return type list of Deployer

4.3.14 bang.deployers.cloud

class bang.deployers.cloud.**BaseDeployer**(*stack*, *config*, *consul*)
Bases: bang.deployers.deployer.Deployer

Base class for all cloud resource deployers

__init__(*stack*, *config*, *consul*)

consul

class bang.deployers.cloud.**BucketDeployer**(**args*, ***kwargs*)
Bases: bang.deployers.cloud.BaseDeployer

__init__(**args*, ***kwargs*)

create()
Creates a new bucket

class bang.deployers.cloud.**CloudManagerServerDeployer**(**args*, ***kwargs*)
Bases: bang.deployers.cloud.ServerDeployer

Server deployer for cloud management services.

Cloud management services like RightScale and Scalr provide constructs like server templates (a.k.a. roles) to bundle together disk image ids with on-server configuration automation (e.g. RightScripts, Scalr scripts). This deployer replaces the low-level provisioning functionality in the base ServerDeployer with a create() method that is more suited to the high-level launching mechanism provided by cloud management services.

__init__(**args*, ***kwargs*)

create()

create_stack()

define()
Defines a new server.

find_def()

class bang.deployers.cloud.**DatabaseDeployer**(**args*, ***kwargs*)
Bases: bang.deployers.cloud.BaseDeployer

__init__(**args*, ***kwargs*)

add_to_inventory()
Adds db host to stack inventory

create()
Creates a new database

find_existing()
Searches for existing db instance with matching name. To match, the existing instance must also be "running".

class bang.deployers.cloud.**LoadBalancerDeployer**(**args*, ***kwargs*)
Bases: bang.deployers.cloud.RegionedDeployer

Cloud-managed load balancer deployer. Assumes a consul able to create and discover LB instances, as well as match existing backend 'nodes' to a list it's given. It is assumed only a single 'instance' per distinct load balancer needs to be created (i.e. that any elasticity is handled by the cloud service).

Example config:

```
load_balancers:
  test_balancer:
    balance_server_name: server_defined_in_servers_section
    region: region-1.geo-1
    provider: hpcloud
    backend_port: '8080'
    protocol: tcp
    port: '443'
```

__init__(**args*, ***kwargs*)

add_to_inventory()
Adds lb IPs to stack inventory

configure_nodes()
Ensure that the LB's nodes matches the stack

create()
Creates a new load balancer

find_existing()
Searches for existing load balancer instance with matching name. Doesn't populate 'details' including the nodes and virtual IPs

class bang.deployers.cloud.**LoadBalancerSecurityGroupsDeployer**(**args*, ***kwargs*)
Bases: bang.deployers.cloud.SecurityGroupRulesetDeployer

__init__(**args*, ***kwargs*)

find_existing()

class bang.deployers.cloud.**RegionedDeployer**(*stack*, *config*, *consul*)
Bases: bang.deployers.cloud.BaseDeployer

Deployer that automatically sets its region

consul

class bang.deployers.cloud.**SSHKeyDeployer**(**args*, ***kwargs*)
Bases: bang.deployers.cloud.RegionedDeployer

Registers SSH keys with cloud providers so they can be used at server-launch time.

__init__(**args*, ***kwargs*)

find_existing()
Searches for an existing SSH key matching the name.

register()
Registers SSH key with provider.

class bang.deployers.cloud.**SecurityGroupDeployer**(**args*, ***kwargs*)
Bases: bang.deployers.cloud.RegionedDeployer

__init__(**args*, ***kwargs*)

create()
Creates a new security group

find_existing()
Finds existing secgroup

class bang.deployers.cloud.**SecurityGroupRulesetDeployer**(**args*, ***kwargs*)
Bases: bang.deployers.cloud.RegionedDeployer

__init__(**args*, ***kwargs*)

apply_rule_changes()
Makes the security group rules match what is defined in the Bang config file.

find_existing()
Finds existing rule in secgroup.

Populates `self.create_these_rules` and `self.delete_these_rules`.

class bang.deployers.cloud.**ServerDeployer**(**args*, ***kwargs*)
Bases: bang.deployers.cloud.RegionedDeployer

__init__(**args*, ***kwargs*)

add_to_inventory()
Adds host to stack inventory

create()
Launches a new server instance.

find_existing()
Searches for existing server instances with matching tags. To match, the existing instances must also be "running".

wait_for_running()
Waits for found servers to be operational

bang.deployers.cloud.**get_deployer**(*provider*, *res_type*)

bang.deployers.cloud.**get_deployers**(*res_config*, *res_type*, *stack*, *creds*)

4.3.15 bang.deployers.default

class bang.deployers.default.**ServerDeployer**(**args*, ***kwargs*)
Bases: bang.deployers.deployer.Deployer

Default deployer that can be used for any servers that are already deployed and do not need special deployment logic (e.g. traditional server rooms, manually deployed cloud servers).

Example of a minimal configuration for a manually provisioned app server:

```
my_app_server:
  hostname: my_hostname_or_ip_address
  groups:
  - ansible_inventory_group_1
  - ansible_inventory_group_n
  config_scopes:
  - config_scope_1
  - config_scope_n
```

__init__(**args*, ***kwargs*)

add_to_inventory()
Adds this server and its hostvars to the ansible inventory.

4.3.16 bang.deployers.deployer

class bang.deployers.deployer.**Deployer**(*stack*, *config*)
Bases: object

Base class for all deployers

__init__(*stack*, *config*)

deploy()

inventory()
Gathers ansible inventory data.

Looks for existing servers that are members of the stack.

Does not attempt to *create* any resources.

run(*action*)
Runs through the phases defined by `action`.

Parameters **action** (*str*) – Either `deploy` or `inventory`.

4.3.17 bang.inventory

class bang.inventory.**BangsibleInventory**(*groups*, *hostvars*)
Bases: ansible.inventory.Inventory

__init__(*groups*, *hostvars*)

get_variables(*hostname*, *vault_password=None*)

is_file()

bang.inventory.**get_ansible_groups**(*group_map*)
Constructs a list of `ansible.inventory.group.Group` objects from a map of lists of host strings.

4.3.18 bang.providers

bang.providers.**get_provider**(*name*, *creds*)
Generates and memoizes a `Provider` object for the given name.

Parameters

- **name** (*str*) – The provider name, as given in the config stanza. This token is used to find the appropriate `Provider`.
- **creds** (*dict*) – The credentials dictionary that is appropriate for the desired provider. Typically, a sub-dict from the main stack config.

Return type `Provider`

4.3.19 bang.providers.bases

class bang.providers.bases.**Consul**(*provider*)
Bases: object

The base class for all service consuls.

Not really the boss of anything, but conveys intent-from-above to foreign entities (e.g. OpenStack Nova/Swift, AWS EC2/S3/RDS, etc...). Also communicates the state of the world back up to the boss.

__init__(*provider*)

class bang.providers.bases.**Provider**(*creds*)
Bases: object

The base class for all providers.

__init__(*creds*)

gen_component_name(*basename*, *postfix_length=13*)
Creates a resource identifier with a random postfix. This is an attempt to minimize name collisions in provider namespaces.

Parameters

- **basename** (*str*) – The string that will be prefixed with the stack name, and postfixed with some random string.
- **postfix_length** (*int*) – The length of the postfix to be appended.

get_consul(*resource_type*)
Returns an object that a Deployer uses to control resources of resource_type.

Parameters **service** (*str*) – Any of the resources defined in bang.resources.

4.3.20 bang.providers.aws

class bang.providers.aws.**AWS**(*creds*)
Bases: bang.providers.bases.Provider

CONSUL_MAP = {'databases': <class 'bang.providers.aws.RDS'>, 'buckets': <class 'bang.providers.a

class bang.providers.aws.**EC2**(**args*, ***kwargs*)
Bases: bang.providers.bases.Consul

The consul for the compute service in AWS (EC2).

__init__(**args*, ***kwargs*)

create_secgroup(*name*, *description*)
Creates a new server security group.

Parameters

- **name** (*str*) – The name of the security group to create.
- **description** (*str*) – A short description of the group.

create_secgroup_rule(*protocol*, *from_port*, *to_port*, *source*, *target*)
Creates a new server security group rule.

Parameters

- **protocol** (*str*) – E.g. `tcp`, `icmp`, etc...
- **from_port** (*int*) – E.g. `1`
- **to_port** (*int*) – E.g. `65535`
- **source** (*str*) –
- **target** (*str*) – The target security group. I.e. the group in which this rule should be created.

create_server(*basename*, *disk_image_id*, *instance_type*, *ssh_key_name*, *tags=None*, *availability_zone=None*, *timeout_s=120*, ***provider_extras*)
Creates a new server instance. This call blocks until the server is created and available for normal use, or `timeout_s` has elapsed.

Parameters

- **basename** (*str*) – An identifier for the server. A random postfix will be appended to this basename to work around OpenStack Nova REST API limitations.
- **disk_image_id** (*str*) – The identifier of the base disk image to use as the rootfs.
- **instance_type** (*str*) – The name of an EC2 instance type.
- **ssh_key_name** (*str*) – The name of the ssh key to inject into the target server's `authorized_keys` file. The key must already have been registered in the target EC2 region.
- **tags** (`Mapping`) – Up to 5 key-value pairs of arbitrary strings to use as *tags* for the server instance.
- **availability_zone** (*str*) – The name of the availability zone in which to place the server.
- **timeout_s** (*float*) – The number of seconds to poll for an active server before failing. Defaults to `0` (i.e. Expect server to be active immediately).

Return type `dict`

delete_secgroup_rule(*rule_def*)
Deletes the security group rule identified by `rule_def`

ec2

find_running(*server_attrs*, *timeout_s*)

find_secgroup(*name*)
Find a security group by name.

Returns a `EC2SecGroup` instance if found, otherwise returns None.

find_servers(*tags*, *running=True*)
Returns any servers in the region that have tags that match the key-value pairs in `tags`.

Parameters

- **tags** (*Mapping*) – A mapping object in which the keys are the tag names and the values are the tag values.
- **running** (*bool*) – A flag to limit server list to instances that are actually *running*.

Return type `list` of `dict` objects. Each `dict` describes a single server instance.

set_region(*region_name*)

class `bang.providers.aws.`**EC2SecGroup**(*ec2sg*)
Bases: `object`

Represents an EC2 security group.

The `rules` attribute is a specialized dict whose keys are the *normalized* rule definitions, and whose values are EC2 grants which can be kwargs-expanded when passing `boto.ec2.securitygroup.SecurityGroup.revoke()`. E.g.:

```
{
    ('tcp', 1, 65535, 'group-foo'): {
        'ip_protocol': 'tcp',
        'from_port': '1',
        'to_port': '65535',
        'src_group': 'group-foo',
        'target': SecurityGroup:group-bar,
        },
    ('tcp', 8080, 8080, '15.183.202.114/32'):  {
        'ip_protocol': 'tcp',
        'from_port': '8080',
        'to_port': '8080',
        'cidr_ip': '15.183.202.114/32',
        'target': SecurityGroup:group-bar,
        },
}
```

This also maintains a reference to the original `boto.ec2.securitygroup.SecurityGroup` instance.

Suitable for returning from `EC2.find_secgroup()`.

__init__(*ec2sg*)

class bang.providers.aws.**RDS**(*provider*)
Bases: `bang.providers.bases.Consul`

class bang.providers.aws.**S3**(**args*, ***kwargs*)
Bases: `bang.providers.bases.Consul`

The consul for the storage service in AWS (S3).

__init__(**args*, ***kwargs*)

create_bucket(*name*)
Creates a new S3 bucket. :param str name: E.g. 'mybucket'

s3

set_region(*region_name*)

bang.providers.aws.**server_to_dict**(*server*)
Returns the `dict` representation of a server object.

The returned `dict` is meant to be consumed by `ServerDeployer` objects.

4.3.21 bang.providers.hpcloud

4.3.22 bang.providers.hpcloud.load_balancer

4.3.23 bang.providers.hpcloud.reddwarf

4.3.24 bang.providers.openstack

4.3.25 bang.providers.rs

4.3.26 bang.stack

class bang.stack.**Stack**(*config*)
Bases: `object`

Deploys infrastructure/platform resources, then configures any deployed servers using ansible playbooks.

__init__(*config*)

Parameters config (`bang.config.Config`) – A mapping object with configuration keys and values. May be arbitrarily nested.

add_host(*host*, *group_names=None*, *host_vars=None*)
Used by deployers to add hosts to the inventory.

Parameters

- **host** (*str*) – The host identifier (e.g. hostname, IP address) to use in the inventory.
- **group_names** (*list*) – A list of group names to which the host belongs. **Note: This list will be sorted in-place.**
- **host_vars** (*dict*) – A mapping object of host *variables*. This can be a nested structure, and is used as the source of all the variables provided to the ansible playbooks. **Note: Additional key-value pairs (e.g. dynamic ansible values like ''inventory_hostname'') will be inserted into this mapping object.**

add_lb_secgroup(*lb_name*, *hosts*, *port*)
Used by the load balancer deployer to register a hostname for a load balancer, in order that security group rules can be applied later. This is multiprocess-safe, but since keys are accessed only be a single load balancer deployer there should be no conflicts.

Parameters **lb_name** (*str*) – The load balancer name (as per the config file)

:param `list` hosts: The load balancer host[s], once known

Parameters **port** – The backend port that the LB will connect on

configure(**args*, ***kwargs*)
Executes the ansible playbooks that configure the servers in the stack.

Assumes that the root playbook directory is `./playbooks/` relative to the stack configuration file. Also sets the ansible *module_path* to be `./common_modules/` relative to the stack configuration file.

E.g. If the stack configuration file is:

```
$HOME/bang-stacks/my_web_service.yml
```

then the root playbook directory is:

```
$HOME/bang-stacks/playbooks/
```

and the ansible module path is:

```
$HOME/bang-stacks/common_modules/
```

deploy()
Iterates through the deployers returned by `self.get_deployers()`.

Deployers in the same stage are run concurrently. The runner only proceeds to the next stage once all of the deployers in the same stage have completed successfully.

Any failures in a stage cause the run to terminate before proceeding to the next stage.

describe()
Iterates through the deployers but doesn't run anything

find_first(*attr_name*, *resources*, *extra_prefix=''*)
Returns the boto object for the first resource in `resources` that belongs to this stack. Uses the attribute specified by `attr_name` to match the stack name.

E.g. An RDS instance for a stack named `foo` might be named `foo-mydb-fis8932ifs`. This call:

```
find_first('id', conn.get_all_dbinstances())
```

would return the boto.rds.dbinstance.DBInstance object whose `id` is `foo-mydb-fis8932ifs`.

Returns None if a matching resource is not found.

If specified, `extra_prefix` is appended to the stack name prefix before matching.

gather_inventory()
Gathers existing inventory info.

Does *not* create any new infrastructure.

get_deployers()
Returns a `list` of *stages*, where each *stage* is a `list` of `Deployer` objects. It defines the execution order of the various deployers.

get_namespace(*key*)
Returns a `SharedNamespace` for the given `key`. These are used by `Deployer` objects of the same `deployer_class` to coordinate control over multiple deployed instances of like resources. E.g. With 5 clones of an application server, 5 `Deployer` objects in separate, concurrent processes will use the same shared namespace to ensure that each object/process controls a distinct server.

Parameters **key** (*str*) – Unique ID for the namespace. `Deployer` objects that call `get_namespace()` with the same `key` will receive the same `SharedNamespace` object.

have_inventory = None
Deployers stash inventory data for any newly-created servers in this mapping object. Note: uses SharedMap because this must be multiprocess-safe.

show_inventory(**args*, ***kwargs*)
Satisfies the `--list` portion of ansible's external inventory API.

Allows `bang` to be used as an external inventory script, for example when running ad-hoc ops tasks. For more details, see: http://ansible.cc/docs/api.html#external-inventory-scripts

bang.stack.**require_inventory**(*f*)

4.3.27 bang.util

class bang.util.**ColoredConsoleFormatter**(*fmt=None*, *datefmt=None*)
Bases: logging.Formatter

format(*record*)

class bang.util.**JSONFormatter**(*config*)
Bases: logging.Formatter

__init__(*config*)

format(*record*)

class bang.util.**NullHandler**(*level=0*)
Bases: logging.Handler

This handler does nothing. It's intended to be used to avoid the "No handlers could be found for logger XXX" one-off warning. This is important for library code, which may contain code to log events. If a user of the library does not configure logging, the one-off warning might be produced; to avoid this, the library developer simply needs to instantiate a NullHandler and add it to the top-level logger of the library module or package.

createLock()

emit(*record*)

handle(*record*)

class bang.util.**S3Handler**(*bucket*, *prefix=''*)
Bases: logging.handlers.BufferingHandler

Buffers all logging events, then uploads them all at once "atexit" to a single file in S3.

__init__(*bucket*, *prefix=''*)

flush()

shouldFlush(*record*)

class bang.util.**SharedMap**(*manager*)
Bases: object

A multiprocess-safe Mapping object that can be used to return values from child processes.

__init__(*manager*)

append(*list_name*, *value*)
Appends value to the list named list_name.

merge(*dict_name*, *values*)

Performs deep-merge of `values` onto the `Mapping` object named `dict_name`.

If `dict_name` does not yet exist, then a deep copy of `values` is assigned as the initial mapping object for the given name.

Parameters **dict_name** (*str*) – The name of the dict onto which the values should be merged.

class bang.util.**SharedNamespace**(*manager*)

Bases: `object`

A multiprocess-safe namespace that can be used to coordinate naming similar resources uniquely. E.g. when searching for existing nodes in a cassandra cluster, you can use this SharedNamespace to make sure other processes aren't looking at the same node.

__init__(*manager*)

add_if_unique(*name*)

Returns `True` on success.

Returns `False` if the name already exists in the namespace.

class bang.util.**StrictAttrBag**(***kwargs*)

Bases: `object`

Generic attribute container that makes constructor arguments available as object attributes.

Checks `__init__()` argument names against lists of *required* and *optional* attributes.

__init__(***kwargs*)

bang.util.**bump_version_tail**(*oldver*)

Takes any dot-separated version string and increments the rightmost field (which it expects to be an integer).

bang.util.**count_by_tag**(*stack*, *descriptor*)

Returns the count of currently running or pending instances that match the given stack and deployer combo

bang.util.**count_to_deploy**(*stack*, *descriptor*, *config_count*)

takes the max of config_count and number of instances running with this stack/descriptor combo

bang.util.**deep_merge_dicts**(*base*, *incoming*)

Performs an *in-place* deep-merge of key-values from `incoming` into `base`. No attempt is made to preserve the original state of the objects passed in as arguments.

Parameters

- **base** (Any `dict`-like object) – The target container for the merged values. This will be modified *in-place*.

- **incoming** (Any `dict`-like object) – The container from which incoming values will be copied. Nested dicts in this will be modified.

Return type None

`bang.util.`**`fork_exec`**(*cmd_list*, *input_data=None*)
Like the subprocess.check_*() helper functions, but tailored to bang.

`cmd_list` is the command to run, and its arguments as a list of strings.

`input_data` is the optional data to pass to the command's stdin.

On success, returns the output (i.e. stdout) of the remote command.

On failure, raises BangError with the command's stderr.

`bang.util.`**`get_argparser`**(*arg_config*)

`bang.util.`**`initialize_logging`**(*config*)

`bang.util.`**`poll_with_timeout`**(*timeout_s*, *break_func*, *wake_every_s=60*)
Calls `break_func` every `wake_every_s` seconds for a total duration of `timeout_s` seconds, or until `break_func` returns something other than `None`.

If `break_func` returns anything other than `None`, that value is returned immediately.

Otherwise, continues polling until the timeout is reached, then returns `None`.

`bang.util.`**`redact_secrets`**(*line*)
Returns a sanitized string for any `line` that looks like it contains a secret (i.e. matches SECRET_PATTERN).

`bang.util.`**`sanitize_config_loglevel`**(*level*)
Kinda sorta backport of loglevel sanitization for Python 2.6.

`bang.util.`**`state_filter`**(*instance*)
Helper function for count_by_tag

CHAPTER 5

Indices and tables

- *genindex*
- *modindex*
- *search*

b

Symbols

A

B

C