test Documentation

Release 1.3.0

webzakimbo

Aug 24, 2020

What is Bcome?

1 About 3

2 Features 5

3 Installation 7

4 Overview 9

5 Basic configuration 11

6 Connecting clouds 13

7 Cleaning up server names 15

8 Caching server lookups 17

9 Inventory filtering 19

10 Sub-selected inventories 23

11 Overview 25

12 Direct connections 27

13 Proxied connections 29

14 Bootstrapping 31

15 Configuration Inheritance 33

16 Overview 37

17 Navigation 39

18 Commands 43

19 Caching Inventories 47

20 Testing connections 49

21 SSH 51

22 Executing commands with run 53

23 Pseudo tty 55

24 File transfers 57

25 Interactive Mode 59

26 Bootstrapping mode 61

27 Port Forwarding 63

28 Bash scripting 65

29 Basic ruby scripting 67

30 The Registry 71

31 The metadata framework 79

32 Internal Hooks 83

33 External Hooks 87

34 Tags 89

Bcome is an orchestration framework for Rubyists: organise your world, then integrate absolutely whatever you want in pure Ruby.

See the project on github.

It's very quick to setup:

It does a lot straight out the box:

And lets you add whatever custom orchestration you require:

(NOTE: Amazon EC2 integration only for the time being. More providers coming).

CHAPTER 1

About

Bcome is a pure Ruby framework delivered as a rubygem, that exposes a rapidly-deployable orchestration toolchain. You can view the project here on github.

Its intention is to stop you having to re-invent the wheel with every project for the most basic of DevOps tasks:

- network discovery
- organising your network's resources in a sensible manner so that you may interact with it holistically
- working out how to access machines and bypass proxies
- writing custom orchestration
- organising the metadata you use in your orchestration, configuration management, and automation processes

1.1 Utility

Bcome allows you to give your developers the tools they need to ship their code faster:

- Share the same tools and scripts between multiple platform simultaneously
- Orchestrate all your platforms at the same time, even if they're in different networks
- Easily plug-in to your existing orchestration
- Install locally as a client-side orchestrator, or remotely as an automation agent
- Avoid duplication and stop doing the same thing over and over for each project
- Code by convention, and concentrate on functionality rather than wiring

1.2 Licensing

Bcome is an open source project, licensed under the GNU General Public License version 3.

The license can be found here: https://github.com/webzakimbo/bcome-kontrol/blob/master/LICENSE.

1.3 Compatibility

Bcome currently integrate with AWS EC2 only, with more providers on the way.

CHAPTER 2

Features

- REPL shell or keyed access from your terminal
- SSH to machines
- Run commands on single or groups of machines
- Upload or download files to single or groups of machines
- Hook in existing orchestration (e.g. deployment or configuration management hooks)
- Hook in custom orchestration in pure Ruby
- Add metadata to enhance orchestration
- Install as a client-side orchestrator or remote automation agent
- Fully extensible & highly-customisable
- Amazon EC2 integration

CHAPTER 3

Installation

3.1 Project structure

First off, you'll need to create your project directory structure

You require a Linux or Unix system with Ruby installed, following which from your command line

Create a directory for your project -

```
mkdir project
```

Within your project directory install the Bcome gem -

```
cd project
project> gem install bcome
```

or add gem 'bcome' to your Gemfile and then -

```
project> bundle install
```

and now create a directory for your Bcome configuration -

```
project> mkdir bcome
```

Within your project directory create an empty yaml file within which your network configuration will live -

```
project> cd bcome
bcome> touch networks.yml
```

Your project directory should look as follows:

```
~> project
  - bcome
    - networks.yml
```

3.2 Cloud integration

Currently Bcome works with a single cloud provider: AWS.

AWS configuration is achieved by linking an AWS IAM user with your local instance of the Bcome client.

3.2.1 Generate an AWS access key and secret access key

Within your AWS account, generate a secret key & secret access key for the IAM user you wish to link to Bcome.

This IAM user should have:

- Programmatic access to the AWS API
- The minimum policy of: AmazonEC2ReadOnlyAccess

Have a look here for an AWS guide on how to do this.

The Bcome framework will use this key & secret in order to conduct queries against Amazon's EC2 API. This allows Bcome to populate your instance with resources from your account.

Note that if you add custom orchestration to Bcome that requires access to features other than EC2, you will of course need to augment the permissions available to your IAM user.

3.2.2 Add your AWS keys to your bcome project

You will need to create a file named .fog in your user's home directory -

```
~/.fog
```

Within this fog file, create a key to reference your AWS account e.g. awsreferencekey

And then within your .fog add in the following yaml:

```
---
awsreferencekey:
   aws_access_key_id: [your access key]
   aws_secret_access_key: [your secret access key]
```

3.2.3 Configuring multiple AWS accounts

Bcome doesn't just work with single AWS accounts - you may configure as many as you like. This allows you to work with machines from disparate accounts from the same project.

This is simple to setup. Given a second AWS account referenced by the key secondawsreferencekey, your .fog file would look as follows:

```
---
awsreferencekey:
  aws_access_key_id: [your access key]
  aws_secret_access_key: [your secret access key]
secondawsreferencekey:
  aws_access_key_id: [second access key]
  aws_secret_access_key: [second secret access key]
```

CHAPTER 4

Overview

To configure Bcome you must define namespaces that represent how you want to group the machines within your estate.

You may have a single platform, representing multiple application environments, or perhaps a large estate comprising multiple application platforms and application environments. You may also want to group machines within your estate by some specific attribute, allowing you to work on only those machines collectively.

Grouping in this way provides you with a means of navigating your estate either within the Bcome shell or by allowing you to key in to specific namespaces - directly from the terminal - in order to directly execute Bcome functions, or any other actions you have defined yourself.

4.1 Namespace types

4.1.1 Collections

A collection may contain other collections, or one or more inventories. A collection could represent a platform, comprising multiple environments, or a collection of platforms.

for example:

- a collection representing a platform, comprising multiple application environments.
- a collection representing an estate, comprising multiple platforms.

It's up to you though what use you make of your collections: they're useful as a namespace to house other things.

4.1.2 Inventories

An inventory contains servers. It could represent all the machines belonging to a specific platform or environment, or a filtered list of machines belonging to a collection

for example:

- an inventory representing a specific application environment, comprising multiple servers.
- an inventory representing all the machines in an given EC2 availability zone, matching certain filters.

Inventories may not contain collections, or other inventories.

Consider an inventory as a collection of servers that you've defined yourself, that you've grouped together because of some shared function.

4.1.3 Sub-selected Inventories

A sub-selected inventory is an inventory that has been further filtered down by specific attributes.

For example:

- a sub-selected inventory comprising only the application servers of a specific type, selected from its the parent inventory.

4.1.4 Servers

A representation of a physical machine, i.e. a server in EC2 or a statically defined server onto which actions may be performed.

CHAPTER 5

Basic configuration

5.1 The network.yml configuration file

Namespaces are defined within the network.yml file in the bcome configuration directory

Navigate to your project directory, and then within the bcome directory, create a networks.yml file

For reference, have a look at what your project structure should look like: *Installation*.

And take a look at what Bcome namespaces are: *Overview*.

5.2 Defining namespaces

Consider the following simple network namespace defined in your networks.yml file:

```
---
"collection1":
  :description: "description of collection 1"
  :type: collection

"collection1:collection2":
  :description: "description of collection 2"
  :type: collection

"collection1:collection2:inventory1":
  :description: "description of inventory"
  :type: inventory
```

This will set up three namespaces

- collection1
- collection1:collection2
- collection1:collection2:inventory1

collection1 contains one collection, collection2, which contains a single inventory, inventory1.

In bcome, the namespacing relationship is defined by a breadcrumb key e.g a:b:c

If we were to add an additional inventory as follows:

```
---
"collection1":
  :description: "description of collection 1"
  :type: collection

"collection1:collection2":
  :description: "description of collection 2"
  :type: collection

"collection1:collection2:inventory1":
  :description: "description of inventory 1"
  :type: inventory

"collection1:collection2:inventory2":
  :description: "description of inventory 2"
  :type: inventory
```

Now collection2 contains two inventories.

Confused? Take a look at the following Asciicast and see how simple it actually is. Note the use of the commands "cd" (change namespace), "ls" (list namespaces), and "tree" (prints a tree view). Note also how bcome can be navigated either via its shell, or by keying directly into your desired namespace.

CHAPTER 6

Connecting clouds

6.1 Connecting your cloud account

How would you add connection details to your cloud provider so that bcome can populate your inventory with servers?

First of all, before starting this section make sure that you've configured your Bcome project for AWS: *Installation*

Let's now assume that our networks.yml configuration is incredibly simple and contains only a single inventory, as follows:

```
---
"inventory1":
  :description: "My inventory"
  :type: inventory
```

We'll now add an EC2 network driver to the inventory.

Let's assume that the credentials for your AWS accounts are keyed on a key called "awsreferencekey", and that you want to retrieve machines from the us-east-1 provisioning region.

Here's what the networks.yml would look like:

```
---
"inventory1":
  :description: "My inventory"
  :type: inventory
  :network:
    :type: ec2
    :credentials_key: awsreferencekey
    :provisioning_region: us-east-1
```

If you've correctly configured your AWS credentials, and have machines in the provisioning region listed, you'll be able to view a list of your servers in that region through bcome:

```
~> bcome ls
```

Let's add another inventory so that we have two inventories, each referencing different EC2 regions.

```
---
estate:
  :type: collection
  :description: "My estate"

  :network:
    :type: ec2
    :credentials_key: awsreferencekey

:estate:inventory1:
  :type: inventory
  :description: "Region 1: us-east"
  :network:
    :provisioning_region: us-east-1

:estate:inventory2:
  :type: inventory
  :description: "Region 2: eu-west"
  :network:
    :provisioning_region: eu-west-1
```

Try and retrieve a machine list for either inventory using bcome:

```
~> bcome inventory2:ls
~> bcome inventory1:ls
```

Note how the network parameters on the collection are inherited in the inventories below, which are then free to override or define as new the provisioning region (or any other key).

All Bcome configuration works in this way: allowing configuration inheritance down your defined namespaces.

CHAPTER 7

Cleaning up server names

You might well be pre-pending your EC2 instance names with a prefix that specifies their context e.g. you may have machines named like this: “Productionapp1”

That’s all well and good, but when browsing in Bcome, the context for your servers should be apparent from how you’ve defined their groupings in your configuration.

You can add the following key at inventory level to your config file:

```
:override_identifier: Production(.+)
```

The override_identifier key takes a regular expression used to clean up your instance names.

The above example would change “Productionapp1” to “app1” within bcome.

See an example of this in the configuration below, where ‘inventory1’ rewrites its servers’ names:

```
---
estate:
  :type: collection
  :description: "My estate"

  :network:
    :type: ec2
    :credentials_key: awsreferencekey

:estate:inventory1:
  :type: inventory
  :description: "Region 1: us-east"
  :network:
    :provisioning_region: us-east-1
  :override_identifier: Production(.+)

:estate:inventory2:
  :type: inventory
  :description: "Region 2: eu-west"
```

(continues on next page)

(continued from previous page)

```
  :network:
    :provisioning_region: eu-west-1
```

The utility of this will become apparent later when you start looking at how to interact with machines.

CHAPTER 8

Caching server lookups

There are two schemes for loading machines into the console: they may be cached, or loaded dynamically each time from the cloud. Although a dynamic load is useful for inventories representing short-lived or often-updated manifests (such as an auto-scaling group), in most cases, caching is preferable.

To cache the servers in a given inventory, enter the console and navigate to your desired namespace, and then hit save. To reload (should you wish to update your manifest), hit reload, and then save again.

```
# caching
> bcome path:to:your:inventory
> save

# reloading
> bcome path:to:your:inventory
> reload
> save
```

When an inventory is cached it will populate a separate config file, machines-cache.yml in your Bcome configuration directory.

Note that you can only cache nodes within an inventory, and not within sub-selected inventories.

Cached inventories will cause a small change to your networks.yml file - your cached inventory will have been marked with the 'load_machines_from_cache' flag, as illustrated in the following example:

```
---
:collection:
  :description: Parent Collection
  :type: collection
  :network:
    :type: ec2
    :credentials_key: youraccount

:collection:useast1:
  :description: Us East 1
  :type: inventory
```

(continues on next page)

(continued from previous page)

```
  :network:
    :provisioning_region: us-east-1
  :load_machines_from_cache: true

:collection:euwest1:
  :description: US East 2
  :type: inventory
  :network:
    :provisioning_region: us-west-1
```

Set this value to false or remove the key to unset caching.

CHAPTER 9

Inventory filtering

9.1 EC2 filter lookups

AWS EC2's full list of lookup-filtering options. may be integrated into your project.

Consider the following simple inventory setup:

```
---
"inventory"
  :description: "My test inventory"
  :type: inventory

  :network:
    :type: ec2
    :credentials_key: "awsreferencekey"
    :provisioning_region: us-east-1
```

Let's add a filter to retrieve just the running instances:

```
---
"inventory"
  :description: "My test inventory"
  :type: inventory

  :network:
    :type: ec2
    :credentials_key: "awsreferencekey"
    :provisioning_region: us-east-1

  :ec2_filters:
    :instance-state-name: running
```

You may add any number of valid EC2 filters to the ec2_filters block in your networks.yml.

This leads to a lot of possibilities as to how you can filter your inventories:

- by VPC
- by architecture
- by instance state
- or by any other or any combination of any allowed filter

Note that if you've cached your server list and you make a subsequent change to your filters, you'll need to reload and then re-save any affected inventory or you won't be able to see your changes.

9.2 Tag-based filtering

Tag-based filtering allows you to apply much more fine-tuning to your inventory manifests than with ec2-filtering alone. This is because tags allow you to apply and then filter on your own domain-specific contexts.

For example, if you were to tag your machines by their function, you would then be able to create inventory manifests by function, and so on.

We cannot express enough how useful tags are in AWS, and highly recommend that you tag your instances.

9.2.1 Visualising your tags within Bcome

Bcome allows you view all the tags added to any given instance.

The 'tags' command when invoked in a server namespace will list out all configured EC2 tags.

```
> bcome collection:inventory:myserver:tags
```

Note also that as for all bcome commands, you may enter the shell and view the tags from there:

```
> bcome collection:inventory:server
> tags
```

9.2.2 Applying tag filters to your network.yml namespaces

Consider a collection containing two inventories, both inventories containing servers from the same ec2 provisioning region as follows:

```
---
:my_application:
  :description: Parent Collection
  :type: collection

  :network:
    :type: ec2
    :credentials_key: awsreferencekey
    :provisioning_region: us-east-1

  :ec2_filters:
    :instance-state: running

:my_application:staging:
   :description: My staging servers
   :type: inventory
```

(continues on next page)

(continued from previous page)

```
:my_application:production:
   :description: My production servers
   :type: inventory
```

The above could represent two different application environments, hosted in the same provisioning regions.

Imagine you have your production servers tagged with a tag named "stage" and a value of "production" in your production environment, and "staging" in your staging environment.

Applying tag-based filters to represent the above scenario would require a configuration as follows:

```
---
:my_app:
  :description: Parent collection
  :type: collection

  :network:
    :type: ec2
    :credentials_key: awsreferencekey
    :provisioning_region: us-east-1

  :ec2_filters:
    :instance-state-name: running

:my_app:staging:
  :description: my staging servers
  :type: inventory
  :ec2_filters:
    :tag:stage: staging

:my_app:production:
  :description: my production servers
  :type: inventory
  :ec2_filters:
    :tag:stage: production
```

Note how tags are just just another type of ec2_filter - the key name being :tag:{[}your tag name]

Note also how :instance-state-name: running is inherited from the parent collection, and is also applied to the inventories below.

The example configuration above would give you two inventories, one returning your production machines, the other your staging machines.

CHAPTER 10

Sub-selected inventories

A sub-selected inventory is a filtered view o another inventory.

Let's take the following simple networks.yml configuration representing a single inventory:

```
---
:myinventory:
  :description: A full list of my servers
  :type: inventory

  :network:
    :type: ec2
    :credentials_key: awsreferencekey
    :provisioning_region: us-east-1

  :ec2_filters:
    :instance-state-name: running
```

We'll expand the example to create a second inventory, sub-selected from the first, where we filter on our EC2 instance tags.

In our example we'll expect a tag called 'function' and we'll supply the filter with an array of values as follows: 'app_server', 'proxy_server':

```
---
:mycollection:
  :description: Collection containing two inventories
  :type: collection
  :network:
    :type: ec2
    :credentials_key: awsreferencekey
    :provisioning_region: us-east-1
  :ec2_filters:
    :instance-state-name: running

:mycollection:myinventory:
```

(continues on next page)

(continued from previous page)

```
  :description: A full list of my servers
  :type: inventory

:mycollection:mysubselect:
  :description: A sub-selected inventory
  :type: inventory-subselect
  :subselect_from: myinventory
  :filters:
    :by_tag:
      :function:
      - app_server
      - proxy_server
```

The above configuration will create a inventory called mysubselect, listing only those servers tagged with a key called 'function', with values of app_server or proxy_server.

Note that when referencing any other namespace in bcome e.g.

```
...
:subselect_from: myinventory
...
```

that the base namespace key (in this case 'my_collection'), is implicit.

Note also that you may subselect from any other namespace, irrespective of where it resides in your overall Bcome namespace scheme.

CHAPTER 11

Overview

All interactions with servers in the Bcome ecosystem are over SSH.

It's a very good idea to have SSH keys in place on the servers with which you're going to interact.

Bcome wil let you -

- execute arbitrary commands on servers or groups of servers (either via the Bcome shell, or directly from your terminal).
- enter an interactive REPL style mode for command execution against individual or groups of servers
- write orchestration scripts
- SSH directly into your servers.
- add in many different sets of SSH credentials, allowing you to apply different connection mechanisms to different namespaces
- work with servers with different connection credentials all at the same time
- work with servers hosted in different cloud accounts, all at the same time
- re-use Bcome's SSH layer for integration with other frameworks

To enable all this you must first configure Bcome for SSH.

11.1 Before you begin

Make sure you understand how namespacing works in Bcome and have tried a few examples: *Overview*

It's also worth taking a look at how to use ping so that you can understand how to verify your configuration: *Testing connections*

CHAPTER 12

Direct connections

Direct connections are when your SSH connections are direct to your instances without going through any interemediary proxy.

You normal means of initiating an SSH connection would look something like this:

```
> ssh user@hostnameorip.com
```

Let's assume you have a single inventory setup, a little like this:

```
---
:myinventory:
  :description: My inventory
  :type: inventory
  :network:
    :type: ec2
    :credentials_key: awsreferencekey
    :provisioning_region: us-east-1

  :ec2_filters:
    :instance-state-name: running
```

Now let's add a basic Direct SSH connection that assumes the following:

- Your SSH user is your local terminal user
- You have ssh keys setup
- Your networks.yml config could look something like this:

```
---
:myinventory:
  :description: My inventory
  :type: inventory
  :network:
    :type: ec2
    :credentials_key: awsreferencekey
```

(continues on next page)

(continued from previous page)

```
    :provisioning_region: us-east-1

  :ec2_filters:
    :instance-state-name: running

 :ssh_settings:
    :ssh_keys:
      - "~/.ssh/id_rsa"
    :timeout_in_seconds: 10
```

To connect as a different user, you can specify the username in the ssh_settings block:

```
---
:ssh_settings:
  :user: "someoneelse"
  :ssh_keys:
    - "~/.ssh/id_rsa"
  :timeout_in_seconds: 10
```

Note the 'timeout_in_seconds' value. This is an integer value representing the time in seconds after which point command execution within Bcome will timeout if a connection cannot be made.

CHAPTER 13

Proxied connections

Proxied connections are where you connect to you instances via some kind of SSH proxy, i.e. through a jump box.

Your normal means of initiating an SSH connection could look something like this:

```
> ssh -o "ProxyCommand ssh -W %h:%p user@jumpboxhost" user@internalhost
```

13.1 by hostname or ip

Let's assume you have a single inventory setup specify your proxy by its hostname or ipaddress:

Your networks.yml would look something like this:

```
---
:myinventory:
  :description: My inventory
  :type: inventory
  :network:
    :type: ec2
    :credentials_key: awsreferencekey
    :provisioning_region: us-east-1

  :ec2_filters:
    :instance-state-name: running

  :ssh_settings:
    :proxy:
      :host_lookup: by_host_or_ip
      :host_id: "xx.xxx.xxx.xxx"
    :ssh_keys:
      - "~/.ssh/id_rsa"
    :timeout_in_seconds: 10
```

To initiate connections using a different jump box user, you would modify your ssh_settings block as follows:

```
...
:ssh_settings:
  :proxy:
    :host_lookup: by_host_or_ip
    :host_id: "xx.xxx.xxx.xxx"
    :bastion_host_user: "someotherusername"
 :ssh_keys:
   - "~/.ssh/id_rsa"
 :timeout_in_seconds: 10
 ...
```

You may also specify a different username for the internal host as follows:

```
...
:ssh_settings:
 :user: "someotherusername"
 :proxy:
   :host_lookup: by_host_or_ip
   :host_id: "xx.xxx.xxx.xxx"
   :bastion_host_user: "someotherusername"
:ssh_keys:
   - "~/.ssh/id_rsa"
 :timeout_in_seconds: 10
```

13.2 by reference to a bcome instance

You can also proxy your SSH connections by reference to another Bcome instance, for example:

```
...
:ssh_settings:
 :proxy:
   :host_lookup: by_bcome_namespace
   :namespace: "inventory:servername"
:ssh_keys:
   - "~/.ssh/id_rsa"
 :timeout_in_seconds: 10
...
```

Note that when specifying a reference Bcome namespace, the highest-level namespace is implicit in the host_lookup declaration.

CHAPTER 14

Bootstrapping

A common requirement is the need to manage machines that have not yet had their configuration applied, i.e. machines that require bootstrapping.

Bcome comes with a bootstrap mode that allows you to define alternative SSH connection configurations.

Consider a simple inventory, as follows, and note the bootstrap_settings block:

```
---
"myinventory":
  :description: "A basic inventory"
  :type: inventory

  :network:
    :type: ec2
    :credentials_key: awsreferencekey
    :provisioning_region: us-east-1

    :ssh_settings:
      :ssh_keys:
        - "~/.ssh/id_rsa"
      :timeout_in_seconds: 10

    :bootstrap_settings:
      :ssh_key_path: path/to/your/private/key.pem"
      :user: username
      :bastion_host_user: ubuntu # optional
```

By default all SSH connectivity would be determined by the ssh_settings block, whilst in bootstrapping mode, SSH connectivity would be determined by the bootstrap_settings block.

See here for how to use bootstrapping from your Bcome shell: *Bootstrapping mode*

CHAPTER 15

Configuration Inheritance

15.1 How inheritance works

Just like the network namespace configuration, SSH configuration is inherited down your namespaces. Lower namespaces are then free to override or define as new the configuration you require.

Take two inventories, the first representing your production servers, and the second, staging. You could define the common aspects of your ssh configuration on the parent collection, which is then inherited by both inventories, and then define different proxy rules on each inventory.

For example:

```
---
:estate:
  :description: Top level collection
  :type: collection

 :network:
   :type: ec2
   :credentials_key: awsreferencekey
   :provisioning_region: us-east-1

   :ssh_settings:
     :ssh_keys:
      - "~/.ssh/id_rsa"
     :timeout_in_seconds: 10

:estate:staging:
  :description: My staging inventory
  :type: inventory

  :ec2_filters:
    :tag:stage: staging

  :ssh_settings:
```

(continues on next page)

(continued from previous page)

```
    :proxy:
      :host_lookup: by_host_or_ip
      :host_id: "111.111.111.11"

:estate:production:
  :description: My production inventory
  :type: inventory

  :ec2_filters:
    :tag:stage: production

  :ssh_settings:
    :proxy:
      :host_lookup: by_host_or_ip
      :host_id: "222.222.222.22"
```

You're free to override the ssh_settings as you see fit to support whichever connection scheme you have setup. For example, you may mix and match Direct Connections or Proxied Connections.

Let's consider a second use case: Two inventories, production and staging, where only production requires proxied access to your hosts:

```
---
:estate:
  :description: Top level collection
  :type: collection

 :network:
    :type: ec2
    :credentials_key: awsreferencekey
    :provisioning_region: us-east-1

  :ssh_settings:
    :ssh_keys:
      - "~/.ssh/id_rsa"
      :timeout_in_seconds: 10

:estate:staging:
  :description: My staging inventory
  :type: inventory

  :ec2_filters:
    :tag:stage: staging


:estate:production:
  :description: My production inventory
  :type: inventory

  :ec2_filters:
    :tag:stage: production

  :ssh_settings:
    :proxy:
      :host_lookup: by_host_or_ip
      :host_id: "xx.xxx.xxx.xxx"
```

15.2 Overriding configuration settings on a per server basis

While by default all your servers will inherit any configuration specified by their parent namespace (as defined in your networks.yml configuration file), it's sometimes useful to override configuration on a per-server basis.

For example, you may have a server deployed within your network that requires different SSH connection parameters. Let's assume that your networks.yml configuration defines your bootstrap SSH user as 'ubuntu' whilst for the server in question, the boostrap ssh user needs to be 'admin'. How would we achieve this?

Let's assume your server is identified by namespace :estate:production:debianserver

15.2.1 The machine-data.yml

Within your bcome config directory, create a file as follows:

```
bcome/machines-data.yml
```

Within your machines-data config, add the following:

```
---
:estate:production:debianserver:
  :ssh_settings:
    :bootstrap_settings:
      :user: admin
```

The bootstrapped SSH user for estate:production:debianserver is now overriden.

You may override any configuration for your servers (or indeed, set any new configuration) using this method.

CHAPTER 16

Overview

All examples within Usage will reference the following pseudo network configuration:

```
root (collection)
  |- staging (inventory)
    |- app1 (server)
  |- qa (inventory)
    |- app1 (server)
  |- production (inventory)
    |- app1 (server)
    |- app2 (server)
```

In the above example we have one root collection named “root”, containing three inventories - staging, qa and production. Staging and qa contain one server each, both named “app1”, whilst the production inventory contains two servers, app1 and app2.

The above example is a common and simple pattern, but there are almost unlimited ways in which you can organise your network. Have a look at the namespace configuration for more information: *Overview*

CHAPTER 17

Navigation

17.1 Basic Navigation

Bcome exposes two methods for navigating through your namespaces:

- direct access via a shell
- keyed accessed to your namespaces directly from the terminal

This section will illustrate the usage of just some of the commands available within the Bcome framework - just enough to get an understanding of how you would invoke them.

To view a list of all commands available within any namespace invoke either 'menu', for in-built Bcome framework commands, or 'registry', for commands you have defined yourself using Bcome's orchestration framework. You can invoke 'menu' or 'registry' for any Bcome breadcrumb (either inside or from outside the shell), e.g.

```
> bcome foo:bar:menu

> bcome foo:bar:registry
```

See the command list section for a full list of available in-built commands.

17.2 Shell access

Bcome exposes a REPL shell, built on top of Ruby's IRB.

From your project directory, you may access it as follows:

```
> bcome
```

This will take you into your root namespace.

To view a tree structure of your namespaces:

```
root> tree
```

List your namespaces:

```
root> ls
```

Enter the context of a second level namespace:

```
root> cd staging
root> staging> ...
```

List the servers present in your staging inventory:

```
root> staging> ls
```

Navigate back up a namespace:

```
root> staging> back
```

Exit out of the shell

```
root> exit!
```

17.3 Keyed access

When you already know the Bcome breadcrumb, keyed access provides a useful shortcut for accessing either common commands, or entering directly into the Bcome shell at a particular namespace.

17.3.1 Keyed access to namespaces

To access the staging namespace directly:

```
> bcome staging
```

Or, given a server named app1 within the production namespace, you'd access it as follows:

```
> bcome production:app1
```

Once within a namespace, invoke either the menu command (for in-built Bcome methods), or the registry command (for user-defined orchestration methods) to find out what you can do next.

17.3.2 Keyed access to commands

Keyed access provides useful shortcuts for common commands.

Any Bcome command, either provided by the framework, or written by you and added to the command registry is available directly from the shell.

For example, to list all the servers in the qa namespace:

```
> bcome qa:ls
```

Or to show all the menu options available for the production namespace

```
> bcome production:menu
```

Or to SSH directly into app1 within the production namespace:

```
> bcome production:app1:ssh
```

Or to access a method you've defined yourself and added to the command registry (in this case a method name foo, available within the context of inventory namespace qa):

```
> bcome qa:foo
```

CHAPTER 18

Commands

18.1 Command list

Slightly different lists of commands are available at collection, inventory and server namespaces.

You may enter Bcome at any namespace and hit menu to view the available command list and usage.

Commands may be grouped into the following:

- Navigation
- Selections
- Server interactions
- Orchestration
- Caching
- Registry

18.1.1 Navigation commands

ls

List all resources available at your current namespace.

cd

Navigate into the namespace of a resource listed within your current namespace.

back

Navigate back to your the namespace.

tree

Print to screen a tree view of all resources below your current namespace.

18.1.2 Selection commands

Bcome allows you to work on selections of resources within any namespace. For example, if your inventory returns 10 servers, and you only want to work with a particular few, you can select these servers only.

You may only work with selections at collection and inventory namespace levels.

workon

Select one or more resources within your current namespace, so that all subsequent commands interact with these selections only.

enable

Add an unselected resource into your current selection.

disable

Remove a selected resource from your current selection.

enable!

All all available resources into your current selection.

disable!

Remove all available resources from your current selection.

lsa

List all resources within your current selection.

18.1.3 Server interaction commands

run

Run allows you to pass a command directly to all servers contained within your current selection. This command will be executed on all servers at all levels including and below your current selection.

For example:

- run within a collection namespace will execute your command on all servers found at all namespace levels below the current collection
- run within an inventory namespace will execute your command on all selected servers (and node: by default, the entire manifest is selected).

- run within a server namespace will execute your command on the server only.

put

Upload a file using SCP to all servers at all levels including and below your current selection.

rsync

Upload a file using Rsync to all servers at all levels including and below your current selection.

interactive

Enter an interactive SSH command session for all servers at all levels including and below your current selection.

ping

Ping all servers to test SSH connectivity.

ssh

SSH directly into a server

inventory and server level only

get

Download a file from a remote server.

inventory level only

18.1.4 Orchestration commands

meta

Output all the metadata configured as part of your orchestration setup for the current namespace. These are made available to your orchestration scripts. See the advanced orchestration section for more information on configuring metadata.

metadata.fetch(:key_name, default)

tags

Output all remote tags configured against a server, e.g. all the EC2 tags. These are made available to your orchestration scripts.

server level only

registry

Show a command list of all user-configured orchestration commands applicable to the current namespace. See the registry section for more information.

18.1.5 Caching commands

save

Cache an inventory locally for faster lookup

inventory level only

reload

Reload an inventory from remote

inventory level only and not available to sub-inventories

18.1.6 Registry commands

The command registry allows you to create your own commands, that will invoke your own custom Ruby orchestration code, executed in the scope of whichever namespace you're currently in.

See our Registry guide on how to create these commands: *The Registry*

Registry commands can either be invoked by keyed access from the terminal

```
> bcome your:namespace:yourcommand
```

Or, they may be invoked from the bcome shell

```
> bcome your:namespace
> yourcommand
```

CHAPTER 19

Caching Inventories

19.1 Overview

You'll notice a slight delay (varying depending on your internet connection) when loading into a Bcome inventory for the first time. This is because Bcome is making a call across the wire to EC2 to retrieve the server manifests matching your particular network configuration and filters.

You may cache these manifests for a speedier loading time.

Caching may only be applied to an inventory namespace, and can not be applied to a collection namespace. This will allow you to leave dynamic inventories (e.g. representing an auto-scaling group or some other scheme where your manifest is subject to regular change) uncached.

Note that examples in this section will utilise the reference network configuration detailed here.

19.2 How to cache

To enable caching, enter the namepace for a given inventory in shell mode and hit save.

e.g. to cache your staging inventory:

```
> bcome staging
root> staging> save
```

You will be prompted to confirm.

Should your server manifest change, you may reload the manifest from remote, and the re-cache as follows:

```
> bcome staging
root> staging> reload
root> staging> save
```

Again, you will be prompted to confirm that you want to proceed

19.3 Caching sub-selected inventories

A sub-selected inventory cannot be directly cached as it is generated as a sub-selection from a parent inventory. You must apply caching to the parent inventory to effect caching on a sub-selection namespace (and in the same vein, all re-caching must be applied to the parent).

19.4 Disabling a cache

When a manifest is cached the following key is added to your network configuration file against the cached manifest:

```
...
  :load_machines_from_cache: true
...
```

To disable a cache, remove this key-value pair entirely or set its value to false, and then re-save your networks.yml network configuration file. You'll need to exit & re-enter any affected Bcome shells you may have open.

CHAPTER 20

Testing connections

SSH Connections may be tested with Bcome's 'ping' command. This can be done either at the level of an individual machine, or groups of machines.

20.1 Ping all machines in an estate

```
> bcome ping
```

20.2 Ping all machines within a given namespace

```
> bcome namespace:ping
```

or

```
> namespace:secondary_namespace:ping
```

20.3 Ping an individual machine

```
> bcome namespace:server:ping
```

20.4 Ping from the Bcome shell

Ping may also be invoked directly from the bcome shell, e.g.

```
> bcome namespace
> cd secondarynamespace
> ping
```

CHAPTER 21

SSH

You may SSH to a server either using keyed-access, or directly from the Bcome shell.

21.1 Using keyed-access

```
> bcome inventory:server:ssh
```

21.2 From the shell

21.2.1 from an inventory namespace

```
> bcome inventory
> ssh servername
```

21.2.2 from a server namespace

```
> bcome inventory
> cd servername
> ssh
```

21.3 See how easy it is

The example below illustrates how Bcome enables SSH to a server behind a proxy.

CHAPTER 22

Executing commands with run

The 'run' command allows you to execute commands on your servers over SSH. You can target either individual, or groups of servers (where you execute the same command on multiple machines in parallel).

The examples in this section will reference the following pseudo network configuration:

```
root (collection)
  |- staging (inventory)
    |- app1 (server)
  |- qa (inventory)
    |- app1 (server)
  |- production (inventory)
    |- app1 (server)
    |- app2 (server)
```

Consider the following use case: You want to view the free memory usage on your servers, and want to use the command "free -m".

22.1 Using keyed access

Execute your command on just app1 within the production namespace:

```
> bcome production:app1:run "free -m"
```

Execute your command on both app1 and app2 within the production namespace:

```
> bcome production:run "free -m"
```

22.2 From the shell

Enter the namespace within which you want to run your command and then enter

```
> run "free -m"
```

If you enter a server namespace, the command will be executed on just that server.

If you enter an inventory namespace, the command will be executed on all selected servers within that inventory.

If you enter a collection namespace, the command will be executed on all servers belonging to all selected inventories & collections within that namespace.

22.3 On a sub-selection of machines

Imagine you want to execute a command on all machines with qa & staging namespaces only, you would:

```
> bcome
> workon qa, staging
> run "command"
```

Or, just 'app1' within the production namespace, you would:

```
> bcome production
> workon app1
> run "command"
```

See the commands 'workon', 'enable', and 'disable' from the commands list for further information on how to filter selections: *Commands*

CHAPTER 23

Pseudo tty

Bcome's pseudo-tty mode allows you to access a pseudo terminal, and is accessible from all server namespaces.

This is useful if you wish to do something like the following:

- Tail a remote log file from your local server
- Open up a remote console, e.g. a MySQL console, Rails console, MongoDb etc

Bcome makes this easy as the SSH connection to your remote machine is already taken care of - you just need to figure out which command you wish to run remotely.

23.1 How to use pseudo-tty within Bcome

23.1.1 Use case 1: tail a remote file

You wish to tail a remote log file, and you usually SSH in to your server and type in the following:

```
> tail -f /path/to/your/file.log
```

Given a server namespace named app1 within a collection namespace of production, you would instead:

```
> bcome production:app1:pseudo_tty "tail -f /path/to/your/file.log"
```

23.1.2 Use case 2: open a mysql console

You wish to open up a Mysql console, and you'd usually SSH in to your server and type in the the following:

```
> mysql -u user -p password -h hostname database
```

Given a server namespace named app1 within a collection namespace of production, you would instead:

```
> bcome production:app1:pseudo_tty "mysql -u user -p password -h hostname database"
```

23.2 Access from the shell

The pseudo_tty function is also accessible directly from the shell.

```
> bcome namespace
> cd server
> pseudo_tty "your command"
```

23.3 Incorporating Pseudo-tty sessions directly into Bcome

The pseudo_tty method is available to all server namespaces within your orchestration scripts too.

This means you can embedd a call to such a function directly into your Bcome installation. For example, I may wish to be able to access a database console directly from Bcome as follows:

```
> bcome staging:app1:db
```

The 'db' invocation would be a a Bcome registry hook, referencing an internal script, within which you can declare the pseudo-tty function as follows:

```
def execute
  @node.pseudo_tty("mysql -u user -p password -h host")
end
```

An example of this in action can be seen below:

To implement something like this for yourself see the following two guides: *The Registry* / *Internal Hooks*

CHAPTER 24

File transfers

Transferring files to and from remote servers can be handled by Bcome.

24.1 get

get allows you to download a file (or recursively download a directory) from a remote server.

Use case: download a file from "/remote/path" on production server app2 and save it to "/local/path"

24.1.1 shell usage

```
> bcome production:app2
root> production> app2> get "/remote/path", "/local/path"
```

24.1.2 Direct usage

```
> bcome production:app2:get "/remote/path" "/local/path"
```

24.2 put

Put allows you to upload a file (or recursively upload a directory) to a remote server, or to a collection of servers simultaneously

24.2.1 Direct upload to an individual server (keyed access)

```
> bcome staging:app1:put "/local/path" "/remote/path"
```

24.2.2 Direct upload to all the servers within a specific inventory

```
> bcome staging:put "/local/path" "/remote/path"
```

24.2.3 Direct upload from the shell, to an individual server

```
> bcome staging:app1
root> staging> app1> put "/local/path", "remote/path"
```

24.2.4 Direct upload from the shell to a server selection

```
> bcome staging
root> staging> put "/local/path", "/remote/path"
```

24.3 rsync

Rsync works exactly like put, but uses Rsync for file transfers rather than SCP.

Rsync is useful when transferring a lot of files as it's quicker.

CHAPTER 25

Interactive Mode

25.1 Overview

Interactive mode allows you to enter repeated commands in a transparent context to either single, or multiple servers without having to enter repeated 'run' commands.

Interactive mode loads a secondary interactive shell, having established an SSH connection to all selected servers, following which any commands you enter will be executed on every server in the selection.

The function is useful when managing groups of servers in a real-time scenario, e.g. applying security patches, testing if servers have a given vulnerability, running updates and the like.

It is expected that those using interactive mode have enough knowledge of the servers under their control, and the repercussions of the commands that they might enter, as all commands entered will be executed in parallel on every server in the selection at once.

Note that the examples in this section will utilise the following reference network configuration:

```
root (collection)
  |- staging (inventory)
    |- app1 (server)
  |- qa (inventory)
    |- app1 (server)
  |- production (inventory)
    |- app1 (server)
    |- app2 (server)
```

25.2 Using interactive mode

Enter the production inventory namespace, and access interactive mode:

```
> bcome production
root> production> interactive
```

You can also access interactive mode directly from your terminal, using keyed access, as follows:

```
> bcome production:interactive
```

You may also want to enter interactive mode for every single server in your estate (something I've done a couple of times when testing for a newly highlighted vulnerability):

```
> bcome interactive
```

25.3 Working with selections of servers

Bcome allows you to work with selections of servers in a given inventory.

Applying this pattern to an interactive session can be very useful should you wish to issue commands to specific groups of servers within a given namespace e.g. "All my application servers within inventory X".

See the Command list of a guide on working with selections: *Commands*

CHAPTER 26

Bootstrapping mode

26.1 Overview

In our configuration section we showed that it's possible to use alternative SSH schemes..

This is known as bootstrapping mode.

It derives from the following common use case: prior to a server being configured, it may be necessary to connect to it using different SSH credentials.

This section details how to use bootstrapping.

See our configuration section for how to configure it: *Bootstrapping*

26.2 Enabling bootstrapping

To enable or disable bootstrapping from the Bcome shell for a given server, enter the namespace for the server in question and enter toggle_bootstrap, e.g. for app1 within inventory namespace qa:

```
> bcome qa:app1
>root> qa> app1> toggle_bootstrap
```

To enable to disable bootstrapping form the Bcome shell for all servers selected witin a namespace:

```
> bcome qa
>root> qa> toggle_bootstrap
```

Bootstrapping mode is also available in your custom orchestration. Given a Bcome namespace held in instance variable @node, you can toggle bootstrapping as follows:

```
@node.toggle_bootstrap
```

All SSH related commands (e.g. SSH'ing to the server, executing run, put, get, rsync, or interactive mode), will use the bootstrapping SSH config when bootstrap mode is enabled.

Note that when interacting with more than one server, one may be running in a bootstrapped context, while the other does not.

CHAPTER 27

Port Forwarding

Bcome allows you to setup local port forwarding with ease - either in bootstrapping mode or otherwise, or for those machines behind a proxy or otherwise.

Let's say you want to forward local port 5901 to destination port 5901 for server namespace :inventory:server

27.1 From the console

```
> bcome inventory:server

# Open a tunnel
estate> inventory> server> tunnel = local_port_forward(5901, 5901)

# Close the tunnel
estate> dev> servera> tunnel.close!
```

27.2 From an orchestration script

Where @node represents your server

```
# Open a tunnel
tunnel = @node.local_port_forward(5901, 5901)

# Close the tunnel
tunnel.close!
```

CHAPTER 28

Bash scripting

28.1 Overview

We've seen already how we can execute single commands in sequence using either the 'run' command or the 'interactive' mode.

Bcome also lets you execute local bash scripts against either individual or groups of your remote servers.

28.2 Setup

The base Bcome directory structure looks as follow:

```
> project
  |- bcome
    |- networks.yml
```

Within the bcome directory, create a new directory called scripts so that your directory structure now looks as follows:

```
> project
  |- bcome
    |- networks.yml
  |- scripts
```

28.3 Hello World

28.3.1 Before we begin

Let's assume you have a network configuration setup as follows - three inventories, within one collection, all containing at least 1 server:

```
root (collection)
  |- staging (inventory)
    |- app1 (server)
  |- qa (inventory)
    |- app1 (server)
  |- production (inventory)
    |- app1 (server)
    |- app2 (server)
```

28.3.2 Create a bash script

Within your scripts directory create the following script hello_world.sh, and into add the following:

```
#!/bin/bash

echo "Hello world, I am `whoami`"
```

28.3.3 Executing your script with keyed access

The quickest & easiest method to execute your script is to use keyed access, where you invoke the execution of your script directly against a known namespace, e.g.

To execute your script on staging, app1:

```
> bcome staging:app1:execute_script hello_world
```

To execute your script on all production servers:

```
> bcome production:execute_script hello_world
```

To execute your script on all servers across your entire estate:

```
> bcome execute_script hello_world
```

Script execution is conducted in parallel across all servers found under your target namespace level.

28.3.4 Executing your script from the shell

First, navigate to your chosen namespace, e.g. for staging, app1

```
> bcome staging:app1
```

And then execute your script

```
root> staging> app1> execute_script "hello_world"
```

Or, to execute against all servers in production:

```
> bcome production
root> production> execute_script "hello_world"
```

You'll notice that in shell mode Bcome returns objects representing the result of the executed command. This will be useful for you later should you wish to incorporate the execution of a bash script into a more advanced Bcome orchestration script.

CHAPTER 29

Basic ruby scripting

29.1 Overview

Bcome lets you write pure Ruby scripts that can interact with your installation. Using this you can:

- interact programmatically with your namespaces and their resources
- access all features of Bcome that are otherwise accessible from the shell or by keyed-access

Basic ruby scripts are not integrated directly into your Bcome framework, and are executed as standalone scripts. Within these scripts you expressely define the namespaces with which you wish to interact.

This will allow you also to integrate Bcome functionality into your other Ruby projects.

29.2 Basics

29.2.1 Network configuration

Let's assume that we have the following basic network configuration:

```
root (collection)
  |- staging (inventory)
    |- app1 (server)
    |- app2 (server)
```

We have a single collection, containing a single inventory, containing two application servers.

29.2.2 Script requirements

At the top of your ruby script, you'll need the following:

```
require 'bcome'

ORCH = Bcome::Orchestrator.instance
# ORCH is an instance of Bcome's Orchestrator that allows us to retrieve namespaces.
```

Let's expand our script to load in the staging inventory:

```
require 'bcome'
ORCH = Bcome::Orchestrator.instance

inventory = ORCH.get("staging")
```

Note that the root namespace is implicit.

If we wanted to load in app1 we would do the following:

```
...
 app1 = ORCH.get("staging:app1")
...
```

29.3 Executing a script

Ruby script execution is the same as for any other Ruby script:

```
> ruby path/to/your/script.rb
```

29.4 Methods

Every method available within Bcome for a given namespace is available to the namespaces retrieved within your scripts (see the command list for a full list).

Remember that to list all integrated Bcome functions for a given namespace, you can invoke the menu function, i.e.

```
require 'bcome'
ORCH = Bcome::Orchestrator.instance
inventory = ORCH.get("staging")
inventory.menu
```

If you've defined your own registry commands, then these will also be available.

You can show your configured registry methods for a given namespace by invoking the registry function, i.e.

```
ORCH = Bcome::Orchestrator.instance
inventory = ORCH.get("staging:app1")
inventory.registry
```

29.5 Accessors

The following accessors are the most useful & commonly used:

29.5.1 For all namespaces

keyed_namespace - returns a string representing the namespace's Bcome breadcrumb

identifier - returns a string representing the namespace's identifier

resources - returns an array of all resources belonging to a namespace, e.g. all the servers belonging to an inventory.

description - the description for the namespace from the network configuration,

type - the namespace type

meta - all configured metadata for this namespace level. See using metadata for more information.

proxy - returns the proxy object associated with a given namespace (if proxied SSH access has been configured)

29.5.2 For sub-selected inventories

parent - returns the parent inventory from which the sub-selected inventory is derived.

29.5.3 For servers

public_ip_address - returns the public IP address, if configured

internal_ip_address - returns the internal IP address, if configured.

ssh_driver.proxy - returns a Ruby Net::SSH::Proxy::Command object if your namespace has a proxy connection configured. This is useful should you wish to re-use your proxy settings in an external script, e.g. within a Capistrano deployment.

tags - returns all configured remote tags. See using tags for more information.

29.5.4 for all inventory types

machine_by_identifier("servername") - load up a server by name from your inventory. Returns a server node.

29.6 Accessing Metadata

The metadata framework makes all metadata available to your ruby scripts.

29.6.1 To list metadata

```
ORCH = Bcome::Orchestrator.instance
inventory = ORCH.get("staging:app1")
inventory.meta
```

29.6.2 To retrieve metadata

```
ORCH = Bcome::Orchestrator.instance
inventory = ORCH.get("staging:app1")
value = inventory.metadata.fetch(:key)
```

See the metadata guide for more information on configuring and using metadata: *The metadata framework*

CHAPTER 30

The Registry

30.1 Introduction

The registry allows you to embed your orchestration scripts within Bcome, so you are able to call them in context either from the shell or as a keyed access command.

By in context we mean that the namespace at which your custom script is called is made available to that script. This allows you to load objects representing your namespaces into the scope of your scripts, at which point you may directly interact with them.

For example, in my own installation I have the following namespaces setup:

```
- ewok (collection)
  - prod (collection)
    - all (inventory)
    - xops (inventory)
```

I have used the registry to provide access to my application deployment scripts from my 'all' inventory, and my operations management scripts from my 'xops' inventory. When I call-up the Registry within Bcome, I get the following:

prod:xops

```
ewok> prod> xops > registry

Registry commands for sub-inventory prod:xops

   deployment

   deploy_foreman  Deploy the network foreman
                   usage: deploy_foreman
                   defaults: build=ewok
                   override: deploy_foreman "build=your-value"


   infrastructure

   generate        generate a cloudformation template
                   usage: generate

   apply           apply the cloudformation stack
                   usage: apply


   puppet

   sync            synchronize puppet manifests up to the puppet master
                   usage: sync

   accept_keys     Accept puppet keys
                   usage: accept_keys

   list_keys       List puppet keys
                   usage: list_keys
```

The 'xops' inventory namespace is made available to the above scripts. Within my scripts I interact with this namespace in order to provide the functionality that my scripts require.

prod:all

```
[ewok> prod> all > registry

Registry commands for inventory prod:all

   deployment

   deploy_pmi       Deploy the pmi application
                    usage: deploy_pmi
                    defaults: build=master
                    override: deploy_pmi "build=your-value"

   deploy_foreman   Deploy the network foreman
                    usage: deploy_foreman
                    defaults: build=ewok
                    override: deploy_foreman "build=your-value"

   deploy_sso       Deploy sso
                    usage: deploy_sso
                    defaults: build=master
                    override: deploy_sso "build=your-value"

   deploy_scorm     Deploy the scorm host application
                    usage: deploy_scorm
                    defaults: build=ewok-release-candidate
                    override: deploy_scorm "build=your-value"

   deploy_skills    Deploy skills index
                    usage: deploy_skills
                    defaults: build=master
                    override: deploy_skills "build=your-value"

   deploy_ready     Deploy marketing readiness
                    usage: deploy_ready
                    defaults: build=master
                    override: deploy_ready "build=your-value"

   deploy_ds        Deploy the death star application
                    usage: deploy_ds
                    defaults: build=master
                    override: deploy_ds "build=your-value"
```

The 'all' inventory namespace is made available to the above deployment scripts. Within these scripts, I use Bcome to determine my deployment targets, and the SSH Proxy I need to traverse in order to push my code. I also use the namespace to pull out any metadata that I require that has been configured against this namespace in Bcome.

Let's move on to how we can configure the registry. Examples in this section will all assume that you have the following Bcome network configuration:

```
root (collection)
  |- staging (inventory)
    |- app1 (server)
  |- qa (inventory)
    |- app1 (server)
  |- production (inventory)
    |- app1 (server)
    |- app2 (server)
```

30.2 The registry.yml configuration file

The base Bcome directory structure looks as follow:

```
> project
  |- bcome
    |- networks.yml
```

Within your bcome directory the registry expects a yaml configuration file named registry.yml, as follows:

```
> project
  |- bcome
    |- networks.yml
    |- registry.yml
```

The yaml configuration is a simple Hash structure referencing arrays of script declarations, keyed on the namespaces to which they are to be made available. The namespace key is a regular expression, allowing you to configure and make available the same orchestration script for different Bcome namespace contexts. For example, the same deployment script made available to all your environments matching a certain Bcome breadcrumb pattern.

```
---
 (regular)expression.+:
   - array
   - of
   - available
   - scripts
 (another|pattern)tomatch?:
   - another
   - list
   - of
   - scripts
```

When navigating within Bcome, you may invoke the registry command, which will list all the available Registry orchestration scripts available to your current namespace, along with usage instructions.

30.3 Registering a script within the Registry

There are three types of Registry orchestration available:

- Shortcuts to commands. You may have commands you invoke regularly on a server that you wish to make available directly from Bcome as a shortcut.
- Internal registry processes that are your own extensions to the Bcome framework. This should be used when you don't have to hand-off your orchestration function to another process.

- External registry processes that are references to external scripts. A common use case here would be a reference to a Capistrano deployment script, where you may pass it the Bcome namespace context, and use Bcome for network discovery and reference your platform metadata.

30.3.1 Registering a shortcut script

Use case 1

I'm using my reference network and wish to create a shortcut for the following command that I normally run on staging:app1

```
> sudo supervisorctl restart unicorn
```

Here's what my Registry could look like:

```
---
"staging:app1":
 - type: shortcut
   description: "Restart the unicorn process"
   console_command: restart_unicorn
   shortcut_command: "sudo supervisorctl restart unicorn"
   group: foobar
```

I now have access to the command restart_unicorn from Bcome context level stag:app1, and I can invoke restart_unicorn either directly from my terminal or from the Bcome shell.

Use case 2

My shortcut command requires a pseudo_tty, for example, I wish to provide a shortcut for tailing a log file. My Registry could look as follows:

```
---
"stag:app1":
 - type: shortcut
   description: "Tail our Nginx access log"
   console_command: tail_nginx
   shortcut_command: "tail -f /var/log/nginx/access.log"
   run_as_pseudo_tty: true
   group: foobar
```

30.3.2 Registering an internal script

Use case 1

I'm using my reference network and have two internal scripts I've prepared earlier. Both manage certain puppet processes within my networks - one synchronises my manifests, and the other accepts my keys. I want to make both pieces of functionality available from Bcome with the commands sync and accept_keys. I want to just apply this to my qa and production environments, and I want to be able to call the commands as follows from my terminal:

~> bcome prod:sync ~> bcome qa:sync ~> bcome prod:accept_keys ~> bcome qa:accept_keys

Here's what my Registry would look like:

```
---
"(qa|production)":
  - type: internal
    description: "synchronize puppet manifests"
    console_command: sync
    group: puppet
    orch_klass: PuppetSync
  - type: internal
    description: "Accept puppet keys"
    console_command: accept_keys
    group: puppet
    orch_klass: PuppetAcceptKeys
```

- description: This is mandatory and will describe the script when it is listed with the registry command from within Bcome
- console_command: The framework will make this command available to you to trigger your script
- group: when listing your available commands within Bcome, the framework will group your commands for you so that they're easier to view.
- orch_klass: This references your Orchestration class. This class is loaded in by the Bcome framework, and is where you place your orchestration code.

See here for a guide on how to write your orchestration classes: *Internal Hooks*

30.3.3 Registering an external script

In configuring an external script, we set our Registry up so that the context within which our script is called may be passed to the external script. Our external script may then load in Bcome and the passed context, and then interact with it.

For example, say I want to make available an external deployment script to my staging inventory. My registry declaration could look as follows:

```
---
"staging":
  - type: external
    description: "Deploy my application"
    console_command: deploy
    group: deployment
    local_command: bundle exec cap myapplication deploy build=%foo%
    defaults:
      build: "master"
```

- description: This is mandatory and will describe the script when it is listed with the registry command from within Bcome
- console_command: The framework will make this command available to you to trigger your script
- group: When listing your available commands within Bcome, the framework will group your commands for you so that they're easier to view.
- local_command: The system command that is to be executed when you invoke your script from Bcome.
- defaults: An array of optional values passed in to your local command. They are useful in that they allow you to pass in additional parameters.

For a guide on how to configure your external script (and for what happens under the hood), see here: *External Hooks*

30.3.4 Listing available registry commands

The 'registry' command will list all your available commands. For a namespace of foo:bar, you may list your commands as follows:

```
> bcome foo:bar:registry
```

CHAPTER 31

The metadata framework

When scripting or otherwise interacting with the Bcome shell the framework lets you access user-defined metadata. This is useful when writing orchestration scripts.

31.1 Defining Metadata

To get started, create a directory called metadata within your project as follows:

```
>project
  |- bcome
    |- metadata/
```

Within the metadata directory, create a yaml file. It doesn't matter what this is called - within the metadata directory Bcome will load up any .yml file it finds, and load them all into memory.

Let's say you have the following reference network configuration:

```
root (collection)
  |- staging (inventory)
    |- app1 (server)
  |- qa (inventory)
    |- app1 (server)
  |- production (inventory)
    |- app1 (server)
    |- app2 (server)
```

And let's say you have a metadata file called mydata.yml. You can begin to add metadata into this file as follows:

```
---
:root:
  value1: some value available to 'root'
  value2: some other value available to 'root'
```

(continues on next page)

(continued from previous page)

```
:root:staging:
  value1: a value available to 'staging' within 'root'

:root:staging:app1:
  value1: a value available to 'app1' within 'staging' within 'root'
```

Above, we have defined metadata for 'root', 'root:staging' and 'root:staging:app1'.

Note that any metadata defined for 'root' will also be made available to all lower namespaces: e.g. ':root:staging' has access to what has been defined in 'root', as does 'root:staging:app1': metadata is inherited down namespaces, and can be overriden.

31.2 Using metadata in scripts

Any Ruby script incorporating Bcome functionality e.g. a basic ruby script, internal or external script, may access Bcome's metadata framework.

Given a Ruby object representing a Bcome namespace, @node, a fetcher is provided:

```
metadata_value = @node.metadata.fetch(:metadata_key)
# OR - you may supply a default value, should the key you request not be defined
metadata_value = @node.metadata.fetch(:metadata_key, {:some => :default }
```

To view what metadata is configured for a given Bcome namespace node, you may instruct Bcome to output the metadata. For example, given namespace foo:bar:

Using keyed access:

```
> bcome foo:bar:meta
```

From the shell

```
> bcome foo:bar
>foo> bar> meta
```

Or within your script, where @node is an instance variable holding a given namespace:

```
puts @node.meta
```

To return an object containing all configured metadata available for a given namespace:

From the shell:

```
> bcome foo:bar
foo> bar> metadata
```

From a script:

```
@node.metadata # for the object wrapper
@node.metadata.data # for a raw hash
```

31.3 Encrypting Metadata

Any metadata files included within your metadata directory may be encrypted with a single key. This allows you to exclude raw metadata files that may contain sensitive information from your source control, and push up encrypted versions instead. The framework will always utilise your encrypted files during its runtime, and will prompt you for your encryption key the first time your encrypted files are required.

31.3.1 Encryption

```
> bcome pack_metadata
```

You'll be prompted for an encryption key. Note that if you re-encrypt your files (e.g. after you've modified your data), your encryption key must match the initial key used to encrypt your files.

Once your metadata is encrypted, Bcome will make us of your encrypted metadata files during its runtime, prompting you for the key the initial time it decrypts your data for use.

31.3.2 Decryption

When you need to work on your metadata files, you may unpack them as follows:

```
> bcome unpack_metadata
```

You'll be prompted for an encryption key.

CHAPTER 32

Internal Hooks

32.1 Introduction

If you haven't already, read up on how the registry allows you to reference your own Ruby extensions to your Bcome framework installation, in the form of internal scripts.

This section deals with how you write these scripts.

32.2 The orchestration directory

Within your bcome project directory, create a directory named orchestration as follows:

```
> project
   |- bcome
     |- networks.yml
     |- registry.yml
   |- orchestration
```

Bcome will expect to find all referenced orchestration scripts within this directory.

32.3 A simple orchestration example

32.3.1 Setup

Let's say you have the following declaration within your registry.yml:

```
---
"foo:bar":
  - type: internal
```

(continues on next page)

(continued from previous page)

```
    description: "synchronize puppet manifests"
    console_command: sync
    group: puppet
    orch_klass: PuppetSync
```

When you invoke foo:bar:sync, Bcome will expect to find a ruby file named puppet_sync.rb within /path/to/your/project/bcome/orchestration/

The puppet_sync.rb needs to look as follows:

```
Module Bcome::Orchestration
  class PuppetSync < Bcome::Orchestration::Base

    def execute
      ... your orchestration code
    end

  end
end
```

Note that your orchestration script file must inherit from Bcome::Orchestration::Base

32.3.2 Accessing your namespace

Within your script, Bcome makes an instance variable named '@node' available to you. This is an instance of your Bcome namespace.

For example, if you invoked the sync command from namespace foo:bar, where 'bar' is an inventory within collection 'foo', then @node will represent your 'foo' inventory.

You can then work with @node to implement your orchestration.

```
Module Bcome::Orchestration
  class PuppetSync < Bcome::Orchestration::Base

    def execute
      # @node = your namespace object
    end

  end
end
```

32.3.3 What can you do with @node?

Every method available within Bcome for a given namespace is available to the @node instance. See the command list for a full list.

A number of accessors are also available to you.

32.4 Passing Parameters

Internal orchestration scripts can also take parameters. This is in the form of a hash, keyed on a variable called defaults, as follows:

```
---
"foo:bar":
  - type: internal
    description: "synchronize puppet manifests"
    console_command: sync
    group: puppet
    orch_klass: PuppetSync
    defaults:
      value1: "foo"
      value2: "bar"
```

From your orchestration scripts, these defaults are accessible from an instance variable named @arguments. For example:

```
Module Bcome::Orchestration
  class PuppetSync < Bcome::Orchestration::Base

    def execute
      # @node = your namespace object
      # @arguments = { :value1 => "foo", :value2 => "bar" }
    end
  end
end
```

As the naming suggests, these parameters are default parameters, and you can override them to pass in different values.

For example, to invoke the above using keyed access from your terminal, and defaulting to the default parameters you would:

```
> bcome foo:bar:sync
```

And to override any of the parameters:

```
> bcome foo:bar:sync value1=your-value
> bcome foo:bar:sync value2=your-value
> bcome foo:bar:sync value1=your-value value2=your-value
```

Remember that if you're ever unsure as to how to invoke your orchestration klass, call up the 'registry' function for your namespace, and your commands and their usage will be shown:

```
> bcome foo:bar:registry
```

Remember also that registry commands may also be triggered from the Bcome shell.

32.5 Invoking an orchestration klass from within another

It is easy to invoke orchestration script from within another.

```
orchestrator = ::Bcome::Orchestration::MyOrchClass.new(node, arguments)
orchestrator.do_execute
```

32.6 Traversing contexts

Although internal scripts are called within the context of a specific namespace available from the @node instance variable, you are not restricted to working solely with this namespace.

For example: you may load in servers from inventory namespaces, or inventories from collection namespace. You may also directly load in unrelated namespaces using the Bcome::Orchestrator class.

See the basic ruby script usage for more information: *Basic ruby scripting*

CHAPTER 33

External Hooks

If you haven't already, read up on how the registry allows you to reference external ruby scripts, and execute them by passing them a Bcome namespace context.

This section will detail with how you write these scripts.

33.1 Setup

The first step is to determine which namespace context you wish to pass to your script.

Let's say that you have a Capistrano deployment script that you normally invoke as follows:

```
> cap myapp deploy
```

And let's say that this you want to apply it to Bcome namespace staging, so that you may invoke your script as follows:

```
> bcome staging:deploy
```

Your registry declaration would look as follows:

```
---
"staging":
  - type: external
    description: "Deploy my application"
    console_command: deploy
    group: deployment
    local_command: cap myapp deploy build=%foo%
```

When you invoke bcome staging:deploy, Bcome will now invoke the following under the hood:

```
> cap myapp deploy
```

Within your myapp Capistrano script, you need to have the following:

```
require 'bcome'
orchestrator = Bcome::Orchestrator.instance
@node = orchestrator.get(ENV["bcome_context"])
```

You script will load in your staging namespace into the @node instance variable, and you may then use Bcome for network discovery i.e. You can setup your target machines by querying Bcome.

You may also make use of Bcome's metadata framework for additional control.

Enabling a call to a script in this way means that:

- your scripts are directly callable from whichever namespace you desire
- you can re-use the same script within another namespace (e.g. for the example given above, you can now re-use the same deployment script in another environment e.g. for a production deploy)
- you can make use of Bcome's metadata framework within your scripts: your scripts may become dynamic according to the namespace from which you have invoked them

Note that every method available within Bcome for your given namespace is available to the @node instance. See the command list for a full list, and also have a look at the accessors.

33.2 Passing parameters to your script and setting defaults

Bcome will also allow you to configure default values to pass to your external script.

Let's expand on the registry configuration we saw earlier:

```
---
"staging":
  - type: external
    description: "Deploy my application"
    console_command: deploy
    group: deployment
    local_command: cap myapp deploy build=%foo%
    defaults:
      build: "master"
```

Now, when you invoke bcome staging:deploy, Bcome will call the following under the hood:

```
> bcome staging:deploy build=master
```

From the command line, you may override any default values as follows:

```
> bcome staging:deploy build=anotherbuild
```

Or from the shell:

```
> bcome staging
staging> deploy "anotherbuild"
```

Any parameters not provided will default to the defaults provided. Note that default values must be provided for all parameters specified.

CHAPTER 34

Tags

Your Tags are the EC2 tags assigned to your instances within AWS EC2 (so unlike metadata, they are available only at server namespaces).

34.1 Listing tags

Let's say you have the following reference network configuration and you wish to view the tags for staging:app1. From your terminal, you would enter:

```
> bcome staging:app1:tags
```

Or from the Bcome shell, you would:

```
 > bcome staging:app1


root> staging> app1> tags
```

34.2 Using Tags inside Bcome scripts

Retrieving tags within your scripts is simple.

Let's assume that your staging:app1 is represented in your scripts by the instance variable @node, and you wish to retrieve a tag named "mytag".

Here's how:

```
tag_value = @node.cloud_tags.fetch(:mytag)
```