Bcome documentation

Release 2.0.0

webzakimbo

Oct 18, 2020

Installation

1 Requirements 3

2 Getting Started 5

3 Installing and Initializing Bcome 7

4 Adding AWS authorization 9

5 Adding GCP authorisation 11

6 Estate 15

7 Network Configuration 17

8 Namespaces 19

9 @node 23

10 SSH 25

11 Overview 27

12 Namespace attributes 29

13 Inheritance & overrides 35

14 Alternative configuration 37

15 Static manifests 41

16 Navigation 43

17 Command Menu 45

18 Executing Commands 49

19 @node methods 51

20 Monkey Patches 53

21 Optimising Ssh 55

22 Ping 57

23 SSH to a server 59

24 Interactive Mode 61

25 Run 63

26 Tunnel 65

27 Pseudo-tty 67

28 The Metadata Framework 69

29 Metadata Encryption 71

30 Metadata Commands 73

31 Bash scripting 75

32 External Scripts 77

33 Internal Scripts 79

34 Registry Overview 83

35 Registry Configuration File 85

36 Registry Configuration Attributes 87

37 Registry method types 89

Welcome to the Bcome Documentation. All you need to configure Bcome to create your DevOps Control Panel.

For functionality demonstrations, please see the Guides.

whats-new-in-2.0.0

breaking-changes-in-2.0.0

CHAPTER 1

Requirements

- Ruby 2.5 or greater.
- A Ruby compatible OS.
- SSH keys configured on your target machines
- A running ssh-agent on your client machine with all relevant ssh keys added. For programmatic access Bcome references your `SSH_AUTH_SOCK` environment variable to find your ssh-agent (and then your keys).
- Servers to connect to !

Note: Bcome uses the `-J` syntax rather than `ProxyCommand` to traverse SSH proxies. As such, Bcome requires an SSH version compatible with the `-J` flag, such as OpenSSH 7.3 or greater, to be installed on the connecting client & proxies.

CHAPTER 2

Getting Started

2.1 Setting up your project

You will need to install and initialize Bcome.

See here for how: *Installing and Initializing Bcome*

2.2 Define your namespaces

Once you've created your project structure the next step is to populate your networks.yml configuration file with your namespace structure.

See *Network Configuration* and *Namespaces* for further information.

2.3 Configure your cloud authorizations

To populate your installation with servers from a cloud provider you will need to add an authorization.

An authorization may be associated with more than one namespace, whilst you may add as many different authorizations as you like to your installation.

When you interact with a Bcome namespace configured for an authorisation, the framework will authenticate you against the cloud provider in question so that it may retrieve a server list, which it will then use to populate your Inventory.

To add an AWS authorization. See: *Adding AWS authorization*.

To add a GCP authorization. See: *Adding GCP authorisation*.

Note: When manually specifying servers you will not need to create an authorisation. See *Static manifests* for further information.

2.4 Add your cloud authorisation to your namespaces

Having added cloud authorisations you will likely want to configure them within your namespaces. This is done by adding to your networks.yml configuration file.

For a full list of networks.yml attributes and their usage see *Namespace attributes*.

Our Guides site is also a useful resource, as it includes example configuration.

2.5 Setup your SSH pathways

SSH is core to Bcome.

If your server instances are not directly accessible you will need to configure your SSH pathways so that Bcome knows how to navigate the proxies you have in place.

This is done by adding to your networks.yml configuration file.

For configuration details, please refer to the SSH Settings Attributes documentation. Our Guides site also has example configuration.

2.6 The Registry & Orchestration

The in-built Registry framework is what makes Bcome your DevOps Console.

It will allow you to associate and re-use custom tasks - i.e. custom orchestration - with your namespaces.

See *Registry Overview* and *Registry method types* for more information. Our Guides site also has example configuration.

CHAPTER 3

Installing and Initializing Bcome

Create a project directory:

```
mkdir project
cd project
```

Install the bcome gem, manually:

```
gem install bcome
```

Or, via a Gemfile:

```
source 'https://rubygems.org'
gem 'bcome'
```

Which you can then install via bundle:

```
bundle install
```

Now run the initializer to create your configuration files & directories:

```
bcome init
```

Your project directory should now look as follows:

```
.
├── .aws
├── .gauth
├── Gemfile
└── bcome
    ├── metadata
    ├── networks.yml
    ├── orchestration
    └── registry.yml
```

CHAPTER 4

Adding AWS authorization

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

If you want to integrate an AWS account, then follow the steps here.

Note: You may connect as many AWS accounts as you like, and mix with as many accounts from other cloud providers.

Bcome will allow you to interact with them all in the same project.

4.1 Create directory structure

Create a directory named `.aws` in the root of your project directory.

If you've correctly setup your project directory structure (see: *Getting Started*), your directory structure should now look like:

```
.
├── .aws
├── Gemfile
└── bcome
    └── networks.yml
```

4.1.1 Generate an AWS access key and secret access key

From within your chosen AWS account, generate a secret key and secret access key for the IAM user you wish to link to Bcome. This IAM user should have:

- Programmatic access to the AWS API
- As minimum, an associated 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: 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.

4.1.2 Add the AWS keys to your bcome project

Create a file named `keys` in your .aws directory

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

And then within your keys file add in the following yaml:

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

Warning: Do not commit your `keys` file to source control.

4.1.3 Configuring multiple AWS accounts

You can add as many AWS accounts as you like. This allows you to work with machines from disparate accounts within the same project.

Given a second AWS account referenced by the key 'my other key', your keys file would look as follows:

```
---
my_key:
  aws_access_key_id: [your access key]
  aws_secret_access_key: [your secret access key]
my_other_key:
  aws_access_key_id: [second access key]
  aws_secret_access_key: [second secret access key]
```

Note: For a demonstration of an AWS authorization in use, please see the AWS EC2 authentication guide

Hint: To add your AWS authorization to your network configuration, see *Network Configuration*.

CHAPTER 5

Adding GCP authorisation

GCP authorisation may be achieved either via *OAuth 2.0*, and/or by directly linking a *Service Account*.

If you want to integrate a GCP account, then follow the steps here.

Note: You may connect as many GCP accounts as you like (using a mix of OAuth 2.0 & Service Account authorisation), and mix with as many accounts from other cloud providers.

Bcome will allow you to interact with them all in the same project.

5.1 Create directory structure

For both `OAuth 2.0` and `Service account` authorisation methods, create a directory named `.gauth` in the root of your project directory.

If you've correctly setup your project directory structure (see: *Getting Started*), your directory structure should now look like:

```
.
├── .gauth
├── Gemfile
└── bcome
    └── networks.yml
```

Warning: Do not commit any files within your .gauth directory to source control. Your OAuth 2.0 & ServiceAccount secrets will live here, as well as any access tokens returned from GCP when OAuth 2.0 is in use.

5.2 OAuth 2.0

To integrate OAuth 2.0 with Bcome, you'll need to create a client id and secret. To do this, follow these steps:

- Login to your GCP web console
- From your projects list select your project (or create a new one)
- Go to APIs & Services
- Go to Credentials
- Select Create Credentials, then select OAuth client id
- Under `Application Type` select `Desktop app` (previously this was 'other')
- Under `Name`, enter a name for your Oauth client application.
- Hit `Create`

Note: If you are prompted to create an OAuth consent screen, you will only need to do so with the minimal required settings of App Name, User Support Email, and Developer Email Address.

Next, make a note of the `Client Id` and `Client Secret` then in your .gauth directory create a file named .gauth/your-secrets-file.json and add the following contents:

```
{
  "installed":
  {
    "client_id": "Your client id",
    "client_secret": "Your client secret",
    "type": "authorized_user"
  }
}
```

If you forgot to make a note of the Client Id and Client Secret, then:

- Login to you GCP web console
- From your projects list select your project
- Go to APIs & Services
- Go to Credentials
- Select your OAuth 2.0 Client application
- Select `Download JSON`

Save this file to your .gauth directory as .gauth/your-secrets-file.json. This file may differ slightly in structure to that suggested above, but it will be compatible.

Note: Your .gauth/your-secrets-file.json can be called anything you like. You'll reference this file later on when you add your authorisation to your network configuration.

Bcome supports multiple GCP authorisations at the same time (either for different GCP accounts, or for different projects within the same account), and you would integrate these by adding a secrets file per GCP project to your .gauth directory.

Warning: Don't commit your secrets file to source control!

As a final step, visit GCP Compute Engine API and hit `ENABLE` to enable the Compute Engine API.

5.3 Service Account

Service Account authorisation requires credentials in JSON format.

- Follow this guide here in order to create your credentials: Creating and managing service account keys
- Download the credentials file in JSON format and save it to your .gauth directory. Your file will look something like this:

```
{
  "type": "service_account",
  "project_id": "your project id",
  "private_key_id": "your private key id",
  "private_key": "your private key",
  "client_email": "your client email",
  "client_id": "your client id",
  "auth_uri": "https://accounts.google.com/o/oauth2/auth",
  "token_uri": "https://oauth2.googleapis.com/token",
  "auth_provider_x509_cert_url": "https://www.googleapis.com/oauth2/v1/certs",
  "client_x509_cert_url": "your client x509 cert url"
}
```

Save your service account credentials json file to your .gauth directory under any name you like. You'll reference this file later on in your networks.yml configuration file.

Note: For demonstrations of GCP authorisation in use, please see our guides: GCP OAuth 2.0 authentication guide / GCP Service Account authentication guide.

Hint: To add your GCP authorisation to your network configuration, see *Network Configuration*.

CHAPTER 6

Estate

The sum-total of all your defined Bcome namespaces is referred to as your `Estate`.

You may construct your estate as follows:

6.1 Single network

A single network from a given cloud provider e.g. a Production environment in AWS.

This is the most basic setup.

6.2 Multi-network

Any number of networks from a given cloud provider e.g. multiple pipeline environments in a single cloud provider that you wish to manage.

This is a typical setup where multiple development & production environments are in play.

6.3 Multi-cloud

Any combination of networks/projects from multiple clouds e.g. a Production environment environment in one cloud, and perhaps a fallback used as part of a disaster recovery system in another.

6.4 Static

On-premise infrastructure, or integrations of providers for which Bcome does not yet have a dedicated driver where your inventories are not dynamically retrieved - rather they are statically defined.

6.5 Hybrid Cloud

Any combination of the above e.g. on-premise infrastructure that you wish to orchestrate alongside cloud-based application environments.

Note: Bcome will allow you to merge all your infrastructure into one installation, such that you may orchestrate across your different clouds & networks at the same time.

CHAPTER 7

Network Configuration

Your Network Configuration defines the architecture of your *Namespaces* and SSH configuration.

In its most basic form it is a single YAML configuration file named `networks.yml` and stored within your `bcome` configuration directory (see: *Getting Started*).

Network Configuration takes the following form:

```
---
namespace_key:
  attribute: value

namespace_key:child_namespace_key
  attribute: value
```

For further information please refer to our *Namespaces* documentation.

The Guides also have example implementations to get you started.

Note: Your network configuration may also be overriden and/or supplemented with additional configuration (see *Alternative configuration*).

CHAPTER 8

Namespaces

Your installation's architecture will be determined directly by how you lay out your namespaces, which should be a direct representation of how you wish to visualize and manage your platforms, environments, applications etc.

The sum-total of all your namespaces is referred to within Bcome as your *Estate*.

Namespaces are declared via YAML (see: *Namespace attributes*) in the following format:

```
---
namespace_key:
  description: "A description of your namespace"
  type:  the namespace type
```

Namespaces are laid out in a parent - child format, for example:

```
---
grandparent:
  description: "The grandparent"
  type: collection

grandparent:parent:
  description: "The parent"
  type: collection

grandparent:parent:child:
  description: "The child"
  type: inventory
```

Note: Namespaces can be of type: `collection`, `inventory`, `inventory-subselect` or `inventory-merge`

8.1 Namespace Types

8.1.1 Collection

Collections may contain any number of other collections and any number of inventories, and is denoted by type 'collection'.

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

Note: Your installation must have at least one, and only one `root` collection. All other namespaces are located below the root. This collection references your Estate, as it contains within it all other namespaces.

8.1.2 Inventory

An inventory contains servers, either populated from a particular cloud account or from a statically defined manifest. It is denoted by type 'inventory'.

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

8.1.3 Sub-selected Inventory

A Sub-selected Inventory contains servers that have been populated as a result of applying filters to an Inventory. It is denoted by type 'inventory-subselect'.

It requires an origin Inventory referenced by the `subselect_from`) attribute upon which the subselect is actioned, in addition to a filter block.

```
---
"root_collection":
  type: collection
  description: My root collection

"root_collection:my_inventory":
  type: inventory
  description: My Inventory

"root_collection:my_sub_selected_inventory":
  type: inventory_subselect
  description: My Sub-selected Inventory
  subselect_from: my_inventory
  filters: ...
```

Note that the 'subselect_from' key denotes a breadcrumb to the subselected from inventory (minus the root collection). For example, consider the following:

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

"estate:platform":
  type: collection
  description: Platform collection

"estate:platform:production":
  type: collection
  description: Production environment

"estate:platform:production:servers":
  type: inventory
  description: All my servers
```

You could create a sub-selection from the 'servers' Inventory as follows:

```
---
"estate:platform:production:app_servers":
  type: inventory-subselect
  description: My application servers for platform production
  subselect_from: platform:production:servers
  sub_filter: {}
```

The root collection key in the sub-select_from attribute - in this instance 'estate' - is implicit.

Note: See: *Namespace attributes* for sub_filter options.

8.1.4 Merged Inventory

A merged inventory is the result of a union of two or more Inventories and/or Sub-selected inventories. It is denoted by type 'inventory-merge'.

It requires contributing Inventories and/or contributing Sub-Selected-inventories to be specified using the 'contributors' attribute. For example:

```
---
"merged:frontend":
  type: inventory-merge
  description: Production and Development frontend servers
  contributors:
  - production:frontend
  - development:frontend
```

Hint: Use Merged Inventories to create 'Multi-cloud' and/or 'Hybrid-cloud' views.

A Merged Inventory may have contributors from the same or different networks within the same Cloud, different Clouds, or from statically declared manifests.

8.1.5 Server

A Server directly represents a server from a Cloud provider, or one from a the statically defined manifest.

With the exception of static-manifests, servers are not declared, rather they are populated into Inventories from by your Network configuration (see: network-configuration-attributes).

CHAPTER 9

@node

`@node` is an instance of a namespace (see: *Namespaces*).

- Whenever you're in a Console session, you are in the scope of an instance of @node.
- When you use Keyed Access to key into a namespace, the last namespace key will reference an instance of @node.
- An instance of @node is made available to you within your orchestration scripts.

@node is the Ruby object that encapsulates the current namespace.

Note: See *Navigation* for the difference between the Console and Keyed-access.

9.1 Interacting with @node

You may directly interact with @node as follows:

- By invoking Menu commands (see: *Command Menu*).
- By invoking custom Registry tasks (see: *Registry Overview*)
- By accessing public-instance methods made available by the framework (see *@node methods*)

CHAPTER 10

SSH

SSH is core to the Bcome framework as all interactions with servers are conducted over SSH.

This is either in the form of programmatic interactions when Bcome is used for orchestration, where Bcome will interrogate your local ssh-agent for your configured keys, or where the framework delegates to your local operating system in order to initiate SSH terminal sessions.

It is HIGHLY recommended that you have SSH keys setup on the machines with which you wish to interact and a running ssh-agent.

Note: Bcome uses the `-J` syntax rather than `ProxyCommand` to traverse SSH proxies. As such, Bcome requires an SSH version compatible with the `-J` flag, such as OpenSSH 7.3 or greater, to be installed on the connecting client & proxies.

Bcome will delegate to your local Operating System for all SSH interactions.

Warning: Ensure that you have your ssh-keys added to your local ssh-agent.

Bcome will reference your ssh-agent's socket via the SSH_AUTH_SOCK environment variable and make use of your configured ssh-keys.

For access to servers where keys are not configured, authorisation will delegate to the installed configuration.

CHAPTER 11

Overview

11.1 Introduction

A Bcome installation is configured through the definition of a networks.yml configuration file, where you -

- Declare namespaces through parent-child relationships
- Declare your namespaces' configuration elements, for example, your SSH architecture.

To begin, navigate to your project directory, and then within the *bcome* directory, ensure that you have created a networks.yml file.

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

Also understand what the available Bcome namespaces are: *Namespaces*.

11.2 Configuration

Your network is configured by first defining *Namespaces*, which determine your installation's architecture, followed by configuring your namespaces by declaring elements.

CHAPTER 12

Namespace attributes

Here you'll find the full list of attributes you may use within your networks.yml file in order to define your *Namespaces*:

12.1 Namespace Block

Used to configure a namespace

attribute key	description	optional	notes
type	Used to define the namespace type.	no	See *Namespaces* for further information Permitted values are 'collection', 'inventory', 'inventory-subselect' and 'inventory-merge'.
description	A description of your namespace.	no	Your description will be used as a label within your installation's UI.
network	A hash of attributes defining a Cloud Provider configuration.	yes	If left blank, any Inventories inheriting this configuration will not be populated with servers unless a Statically defined manifest has been configured Restricted to namespaces of type 'collection' and 'inventory' only. See *Network attributes*.
ssh_settings	A hash of attributes used to define an SSH architecture.	yes	Leave this blank and Bcome will default to initiating direct SSH connection attempts only (i.e. no proxies) and will fallback to using your terminal user as your SSH username. Restricted to namespaces of type 'collection' and 'inventory' only. See *SSH Settings Attributes*.
sub_filter	A hash of attributes used to further filter a list of machines from an inventory.	yes	Restricted to namespaces of type 'inventory-subselect' only. If you're sub-filtering a 'gcp' inventory, your filters are a Hash of GCP tags and their values. If you're sub-filtering an 'aws' inventory, your filters are a Hash of EC2 and their values.
override_identifier	A regular expression used to rewrite the names of servers within an inventory	yes	Restricted to namespaces of type 'inventory', 'inventory-subselect' and 'inventory-merge'. A regular expression with a single selector is expected, for example given a server named "Foo_Bar" and a regular expression of "[a-z]*_([a-z]*)" the server will be renamed "Bar".
hidden	A toggle to hide a namespace from view.	yes	set to 'true' or 'false' Hidden namespaces may still be interacted with, but will not appear in the user interface.

Note: Note that `ssh_settings` and `network` configuration may be inherited and overidden in child namespaces.

See *Inheritance & overrides*

12.1.1 Network attributes

A Hash of attributes used to populate the top-level `network` attribute.

Used to configure a Cloud-provider.

See the full list of configurable attributes here:

attribute key	description	optional	notes
type	The cloud provider type	yes	Set to "gcp" for Google Cloud Platform. Set to "ec2" for Amazon Web Services.

Google Cloud platform specific network attributes

attribute key	description	optional	notes
project	GCP project id	Required for 'gcp' provider type	Be careful to set this to the project id, and not the project name.
zone	GCP zone	Required for 'gcp' provider type	For a full list of zones see: Zones & Clusters.
authentication_scheme	GCP authentication scheme	Required for 'gcp' provider type	Supported schemes are 'oauth' or 'service_account'. For OAuth 2.0 setup see *Adding GCP authorisation*
service_scopes	An array of GCP auth scopes passed to GCP during authorisation.	Optional for 'gcp' provider type	A minimum scope of compute.readonly is required in order to list resources. For OAuth 2.0 cloud-platform is required.
filters ('gcp' provider)	A filter string to filter instances returned by GCP.	Optional for 'gcp' provider type	As an example, to return running instances, set filter to "status:running" For further information on topic filtering, see GCP Topic Filtering.
service_account_credentials	The name of the service account credentials json file, to be found within the .gauth directory.	Optional for 'gcp' provider type	Required for the service_account authentication scheme only. See *Adding GCP authorisation*.
secrets_filename	The name of your OAuth 2.0 clients secrets filename to be found within the .gauth directory.	Optional for 'gcp' provider type	Required for the oauth authentication scheme only. See *Adding GCP authorisation*.

Note: Google Cloud Platform require a minimum permission of `compute.instances.list` for OAuth 2.0 authorisations. Ensure that any users attempting to authorize by OAuth 2.0 have been configured with a role containing this permission.

AWS specific network attributes

attribute key	description	optional	notes
credentials_keys	The reference to an AWS credentials key from your .aws/keys file	Required for 'ec2' provider type	For setup see *Adding AWS authorization*.
provisioning_region	An EC2 provisioning region	Required for 'ec2' provider type	e.g. eu-west-1
filters ('ec2' provider)	A hash of ec2 filters sent during the lookup request to ec2.	Optional for 'ec2' provider type	For a full list of available filters, see EC2 Filter List.

12.1.2 SSH Settings Attributes

A hash of attributes used to populate the top-level `ssh_settings` attribute.

Note: Namespaces without an SSH Settings element will default to initiating direct connection attempts against your servers using your local terminal username as the SSH username. Proxied ssh connections will not be possible.

See the full list of configurable attributes here:

attribute key	description	optional	notes
user	The SSH username to use for SSH connections.	Yes	Most implementations will leave this blank, causing Bcome to fallback to using the local terminal user's username as the SSH user. Setting a username within will override this.
proxy	An array of proxies	Yes	If proxies are configured, Bcome will craft an SSH connection jumping through each proxy in the order in which they are declared in the proxy array, position 0 being first hop. If a proxy is a Bcome node, its public_ip_address will be used to route the connection if present. If there is no public_ip_address available, Bcome will default to using the node's internal_ip_address to route the connection. Using this pattern you may proxy via multiple hops into your networks. See *Proxy Attributes*.

Proxy Attributes

Used to define an SSH Proxy.

See the full list of configurable attributes here:

attribute key	description	optional	notes
host_lookup_type	The type of host lookup to perform.	No	Permitted values are: 'by_bcome_namespace', 'by_host_or_ip' or 'by_inventory_node'. Note that 'by_host_or_ip' must be used to reference proxies without public interfaces. A future release will enable such lookups using *by_bcome_namespace*.
namespace	A bcome namespace in breadcrumb format, e.g. namespace_key:namespace_key	Yes	Required for host_lookup type 'by_bcome_namespace'. Allows for referencing proxy machines that can be defined anwywhere within the Bcome installation.
host_id	A hostname or ip address, or reference to a host from your ssh config or hosts file. In other words, anything that your underlying OS can resolve as an SSH target	Yes	Required for host_lookup type 'by_host_or_ip'.
node_identifier	The name of the node within the same Inventory that you wish to declare as your SSH proxy machine.	Yes	Required for host_lookup type 'by_inventory_node'.
bastion_host_user	The ssh username to be used for bypassing the proxy.	Yes	Default to the Bcome installation's local SSH username.

CHAPTER 13

Inheritance & overrides

Network and Ssh Configuration (see *Namespace attributes*) are inherited by all of a namespace's children, at which point they may be overidden to customise the attributes at that namespace level.

For example:

```
---
"estate":
  type: collection
  description: My Estate
  network:
    foo: value
    bar: other value

"parent:child":
  type: inventory
  description My Inventory
```

The 'child' element's network attributes looks as follows:

```
--
network:
  foo: value
  bar: other value
```

Then consider:

```
---
"estate":
  type: collection
  description: My Estate
  network:
    foo: value
    bar: other value
```

(continues on next page)

(continued from previous page)

```
"parent:child":
  type: inventory
  description My Inventory
  network:
    bar: overriden value
```

The child element's network Element has been partially overriden resulting in the following:

```
--
network:
  foo: value
  bar: overriden value
```

CHAPTER 14

Alternative configuration

14.1 Overriding network.yml with CONF=

If you want to use an alternative network configuration to the default file at

```
.
├── Gemfile
└── bcome
    └── networks.yml
```

then create your new configuration, e.g.

```
.
├── Gemfile
└── bcome
    └── networks.yml
    └── alternative-configuration.yml
```

To use your alternative file, set a reference to it using the CONF= environment variable when invoking Bcome, for example:

```
CONF=bcome/alternative-configuration.yml bcome command
```

Hint: Using the CONF environment variable is useful if you want to support different views of your infrastructure (e.g. providing views to teams based on role), or if you want to provide access to multiple installations from the same project.

14.2 Ad-hoc overriding of network.yml with ME=

You may already have seen how configuration may be inherited and overidden in Bcome (see *Inheritance & overrides*).

This can be also be done by referencing an overrides configuration file, and referencing it using the ME environment variable.

For example, consider the following Yaml configuration:

```
---
foo:bar:
  ssh_settings:
    user: "ubuntu"
```

Save it to the bcome/ configuration directory:

```
.
├── Gemfile
└── bcome
    └── networks.yml
    └── username-override.yml
```

You may now invoke it as follows:

```
ME=bcome/username-override.yml bcome command
```

In the example given above this particular configuration override will override the ssh username for namespace "foo:bar". You may override any configuration in this way.

This can be useful for

- Supporting a temporary configuration of an installation, for example in order to orchestrate a server installed with a bare operating system pre-bootstrapping, where only the default system users are present.
- Providing a local override for attributes

Note: You can use both CONF= and ME= at the same time. CONF will load an alternative networks.yml configuration file, and ME will provide overrides.

14.3 The me.yml configuration file

As well as setting an override using ME=, you may place the same contents in a file named me.yml, as follows:

```
.
├── Gemfile
└── bcome
    └── networks.yml
    └── me.yml
```

Contents in me.yml will be automatically loaded and used to override your Network configuration.

Hint: This is the best way to provide your own personal override within collaborative projects.

14.4 Overriding an individual server's configuration

To override an individual server's configuration, you must use a server-overrides.yml configuration file:

```
.
├── Gemfile
└── bcome
    └── networks.yml
    └── server-overrides.yml
```

For example, to override the connection details for 'server_a' within namespace 'foo:bar', your server-overrides.yml configuration would look as follows:

```
---
foo:bar:server_a:
  ssh_settings:
    user: a_different_username
```

Note: Any server-specific overrides must be placed within the server-overrides.yml override file, and cannot be placed in the general or overriden network configuration.

CHAPTER 15

Static manifests

15.1 Overview

If you have infrastructure on-site, or in any other provider for which Bcome does not yet provide a driver, you may utilise a Static Manifest.

A Static Manifest is a list of manually configured servers that can be used to populate a specified Inventory.

For example, consider the following simple Network configuration:

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

estate:onsite:
  type: inventory
  description: My onsite inventory
```

Above we have defined a Collection housing a single Inventory (see *Namespaces*). There is no Network defined (see: *Namespace attributes*), and therefore no configured cloud provider to populate the Inventory.

Now create a file called `static-cache.yml` and save it to your Bcome configuration directory as follows:

```
.
├── Gemfile
└── bcome
    └── networks.yml
    └── static-cache.yml
```

Here's a simple example of a Static Manifest entry within our static-cache.yml file that would populate our Inventory with two servers:

```
---
estate:onsite:
```

(continues on next page)

(continued from previous page)

```
- identifier: file_server_one
  public_ip_address: 123.123.123.12
  description: My server
  cloud_tags:
    data:
      a_key: a_value
      another_key: another_value
- identifier: some_other_server
  public_ip_address: 678.678.678.67
  internal_ip_address: 10.2.23.12
  description: My other server
```

15.2 Attribute List

attribute key	description	optional	notes
identifier	The server name.	No	Bcome will automatically swap whitespace for underscores, and auto-increment duplicate identifiers. A server's identifier is incorporated into its namespace breadcrumb.
description	The server description	No	A description of the server. This will appear in Bcome's UI.
public_ip_address	The public interface IP address.	Yes	You may use a hostname here also.
internal_ip_address	The internal interface IP address.	Yes	You may use a hostname here also.
local_network	Set to true or false. Indicates whether the server is to be found on the local network.	Yes	If a set with local_network: true, Bcome will initiate SSH connections on the internal_ip_address. If set to false, connections will fallback to proxy (if configured in the namespace's network configuration) or to the public_ip_address.
cloud_tags	A Hash of tags keys and values, keyed on :data	Yes	See *Tag attributes* for structure

15.2.1 Tag attributes

Cloud tag attributes have the following YAML structure:

```
---
cloud_tags:
  data:
    tag_key_1: tag_value_1
    tag_key_2: tag_value_2
```

CHAPTER 16

Navigation

Bcome is used by navigating to a namespace, and invoking the method of your choice.

That method may either be a built-in function (see: *Command Menu*), or a custom function you've added yourself as part of your installation, and available via the Registry (see *Registry Overview*).

You may either traverse your namespaces via *The Console* before invoking your method, or you may invoke it directly from your terminal using *Keyed-Access*.

Let's imagine you have *Namespaces* laid out in the following parent-child relationship:

```
.
└── parent
    └── child
        └── grandchild
```

16.1 The Console

Bcome exposes a REPL (read-eval-print loop) shell, built on top of Ruby's IRB (interactive Ruby) shell.

Each namespace is loaded into a distinct shell session, which you may then interact with directly.

16.1.1 Basic navigation

Enter the Console at the root namespace:

```
> bcome
```

List your namespaces:

```
> ls
```

Traverse to the 'child' namespace, and onward to 'grandchild':

```
> cd child
> cd grandchild
```

Go back up a level:

```
> back
```

Exit back to you terminal:

```
> exit!
```

Hint: For a full list of in-built commands see menu.

For accessing your custom commands, see the Registry (*Registry Overview*).

16.2 Keyed-Access

Bcome provides a shortcut to any namespace, referred to as Keyed-Access, where the namespace breadcrumb is included as a parameter to `bcome`.

16.2.1 Enter the Console using Keyed-Access.

To enter the CLI directly at our 'grandchild' namespace, you would enter the following command:

```
> bcome child:grandchild
```

This allows you to start a Console session directly at the namespace you require, without having to traverse your tree.

Hint: Invoking 'ls' on any namespace will list its direct children, whilst invoking 'tree' on any namespace will list the Bcome tree structure beneath.

See *Executing Commands* for how to execute commands in Keyed-access or Console mode.

CHAPTER 17

Command Menu

Bcome has a series of in-built commands that can be executed within your namespaces.

Within any given namespace, invoke `menu` to pull up the available in-built commands.

Note: When in Keyed-Access mode (see: *Navigation*) command parameters are whitespace rather than comma-delineated as they are for Console mode.

For a guide to executing commands, see *Executing Commands*.

All in-built commands are described below:

17.1 Command lists

command	description	namespace availability
menu	Returns a list of all the in-built commands available at the current namespace.	All
registry	Returns a list of all the custom commands configured to be available at the current namespace.	All

17.2 Navigation commands

command	description	namespace availability
cd *identifier*	Enter a console session for a child namespace. Console only	Collection, Inventory, Sub-selected Inventory, Merged Inventory.
back	Return to the parent namespace console session. If at the root namespace, or within the namespace at which the Console was initiated, invoking 'back' will exit the Console. *Console only*	All
exit	see 'back', above. *Console only*	All
exit!	Exits a Console session. *Console only*	All

17.3 Selection Commands

command	description	namespace availability
workon *identifier1*, *identifier2* ...	Work on specific namespaces only, inactivating all others from the selection. *Console only*	Collection, Inventory, Sub-selected Inventory, Merged Inventory.
enable *identifier1*, *identifier2* ...	Re-enable a namespace within the namespace. *Console only*	Collection, Inventory, Sub-selected Inventory, Merged Inventory.
disable *identifier1*, *identifier2* ...	Remove a namespace from the selection. *Console only*	Collection, Inventory, Sub-selected Inventory, Merged Inventory.
enable!	Enable all namespaces within the selection. *Console only*	Collection, Inventory, Sub-selected Inventory, Merged Inventory.
disable!	Disable all namespaces within the selection. *Console only*	Collection, Inventory, Sub-selected Inventory, Merged Inventory.
first	A shortcut available in Keyed Access mode only when traversing namespaces. e.g. bcome foo:bar:first:command *Keyed-access only*	Collection, Inventory, Sub-selected Inventory, Merged Inventory.

17.4 SSH Commands

command	description	namespace availability
ping	Test connectivity against all servers in the selection.	All
run *command1*, *command2*	Execute a command to be run against *all* servers contained in the selection.	All
interactive	Access an interactive ssh pseudo-shell allowing you to execute commands against all servers that are children or grandchildren of the current namespace.	All
tunnel *local port*, *destination port*	Create a Tunnel over SSH Transparently bypasses proxies.	Server.
pseudo_tty *command*	Invoke a pseudo-tty session.	Server
execute_script *path/to/bash/script*	Execute a local bash-script over ssh against all servers in the selection.	All
ssh *identifier*	Ssh to a server	Inventory, Sub-selected Inventory, Merged Inventory.
ssh	Ssh to a server	Server

17.5 File & script commands

command	description	namespace availability
put *local/path*, */remote/path*	Upload a file (or directory, recursively) over SCP to all servers in the selection.	All
rsync *local/path* */remote/path*	Upload a file (or directory, recursively) over rsync (faster) to all servers in the selection.	All
put_str *"string contents"* */remote/path*	Write a file /to/remote/path to all servers in the selection from a String.	All
get */remote/path*	Download a file or directory.	Server

17.6 Informational

command	description	namespace availability
ls	List all child namespaces. *Console only*	Collection, Inventory, Sub-selected Inventory, Merged Inventory.
lsa	List all active child namespaces See *Selection Commands* *Console only*	Collection, Inventory, Sub-selected Inventory, Merged Inventory.
tree	Print a tree view of all child namespaces and their sub-namespaces.	Collection, Inventory, Sub-selected Inventory, Merged Inventory.
meta	List all Metadata available to the current namespace. See: :doc: *../metadata/metadata-commands* for working with metadata.	All
tags	List all cloud provider tags/labels associated with the server(s)	Server, Inventory, Sub-selected Inventory, Merged Inventory.
routes	Print SSH routing tree.	All

17.7 Miscellaneous

command	description	namespace availability
reload	Re-populate an inventory from its source, e.g. reload all servers from within an inventory from its remote cloud.	Inventory, Sub-selected Inventory
cache	Save a local copy of an inventory. The cache is saved as a static manifest. (see: *Static manifests*) To disable the Cache, remove the static manifest.	Inventory.

CHAPTER 18

Executing Commands

Let's imagine you have *Namespaces* laid out in the following parent-child relationship:

```
.
└── parent
    └── child
        └── grandchild
```

18.1 Invoking a command when in the console

Given command 'foo' available to namespace parent:child:grandchild, how would you invoke it?

```
> bcome child:grandchild
> foo
```

Note: When you an enter an IRB session, you are in the scope of an instance of a Ruby object (see: *@node*).

Bcome's Console assigns an instance of your namespace as this session object.

18.2 Invoking commands with Keyed-access

To invoke command 'foo' available to namespace parent:child:grandchild, you would enter the following command in your terminal:

```
> bcome child:grandchild:foo
```

18.3 Invoking commands from orchestration scripts

If you have a namespace instance, for example *@node*, then any available command may be invoked directly on it.

For example, given a namespace in an instance var @node, and a command, :foo, you would:

```
@node.foo
```

Hint: Any command, be they in-built menu commands (see *Command Menu*), public-methods on a given namespace object, or a custom-command you've defined yourself in the Registry (see *Registry Overview*), is accessible in this way.

CHAPTER 19

@node methods

Various public-instance methods are made available on the *@node* object.

These are useful for debugging in Console sessions and for enhancing your orchestration scripts.

Notable accessors are detailed below:

19.1 On all namespaces

19.1.1 Networking accessors

The network driver

```
network_driver
```

Its configuration, as a Hash:

```
network_driver.config
```

The authenticated Cloud credentials for the current namespace, returned as a Hash. For EC2 this will return the namespace's access key and secret key, for GCP it will return an access_token and the project name. This is useful should you wish to extend an authenticated Bcome session into a custom integration.

```
network_driver.network_credentials
```

The SSH driver, an instance of `::Bcome::Ssh::Driver`

```
ssh_driver
```

The `Net::SSH::Proxy::Jump` configuration, should the namespace be configured to proxy its connections:

```
ssh_driver.proxy
```

19.1.2 Server accessors

All servers present within the current namespace:

```
machines
```

19.1.3 Metadata accessors

The metadata object for this namespace (`::Bcome::Node::Meta::Local`):

```
metadata
```

All metadata for this namespace as a Hash

```
metadata.all
```

19.2 On server namespaces

The cloud server if @node is an EC2 server (returns an instance of Fog::Compute::AWS::Server):

```
ec2_server
```

The cloud server if @node is a GCP server (returns an instance of Google::Apis::ComputeBeta::Instance):

```
gcp_server
```

Return all configured tags/labels if a cloud server:

```
tags_h
```

Find cloud tags/labels value by key:

```
cloud_tags.fetch(:tag_or_label_name, "optional_default_value")
```

Note: @node may also be extended by applying *Monkey Patches*.

CHAPTER 20

Monkey Patches

The Bcome framework may be extended by applying a monkey patch.

To create a patch ensure you have a 'patches' directory, as follows:

```
.
└── project_directory
    └── bcome
        └── patches
```

Any Ruby file placed into 'patches' with a .rb extension will be loaded into the framework.

As an example, to add a method named 'foo' that returns 'bar' onto a GCP server, create a file called my_patch.rb and place it into the patches directory:

```
.
└── project
    └── bcome
        └── patches
            └── my_patch.rb
```

Within it add the following code:

```ruby
class Bcome::Node::Server::Dynamic::Gcp
  def foo
    puts "bar"
  end
end
```

All GCP server instances would now be patched with the new 'foo' method.

See here for Bcome's github source.

CHAPTER 21

Optimising Ssh

When interacting with machines over SSH Bcome will attempt to connect to all machines in any given selection at once, re-trying connection attempts and auto-reconnecting should connections be lost.

It will also cache the connections until the Bcome session has terminated.

This can be problemmatic for some jump hosts with default configuration, that may not be setup to handle the volume of concurrent connection requests that Bcome will make (one per server in your selected namespace).

This can be resolved by correctly setting your jump host's SSH daemon's `MaxStartups` and `MaxSessions` attributes (if you're running OpenSSH) in line with your installation's requirements.

Note: As a general rule of thumb set MaxStartups and MaxSessions to be equal to the number of machines you need to manage via a particular jump host. A lower number will result in dropped connections, and although Bcome will catch these and re-connect, this can result in a slower startup time.

CHAPTER 22

Ping

22.1 Overview

SSH Connections may be tested with Bcome's `ping` command.

For each server in scope the connection will be tested and a status (containing the server's connection configuration) returned.

The returned status will look like this:

```
"your_estate:an_inventory:some_server" => {
  "connection" => "success",
  "ssh_config" => {
      :user    => "guillaume",
      :timeout => 1,
      :proxy   => [
        [0] {
          :host_lookup => "by_bcome_namespace",
          :namespace   => "your_inventory:a_jump_host",
          :proxy_host  => "12.345.678.90",
          :user        => "guillaume"
        }
      ]
    }
  }
}
```

Note: Failed connections will be coloured red, and successful ones green. Each is marked with a connection status of "failed" or "success" respectively.

22.2 Usage

Let's imagine you have *Namespaces* laid out in the following parent-child relationship:

```
.
└── parent
    └── child1
    └── child2
```

And let's assume that 'child1' and 'child2' are inventories containing servers.

22.2.1 Ping all the servers within your Estate

```
bcome ping
```

22.2.2 Ping all the servers within a given inventory

```
bcome child1:ping
```

22.2.3 Ping an individual server within a given inventory

```
bcome child2:your_server:ping
```

22.2.4 Ping from the Bcome Console

```
bcome child1
ping
```

Note: Bcome caches all SSH connections, with the exception of those made during a Ping.

CHAPTER 23

SSH to a server

You may SSH to a server either using Keyed-Access, or directly from the Bcome console (see: *Navigation*).

Let's imagine you have *Namespaces* laid out in the following parent-child relationship:

```
.
└── parent
    └── child
        └── server
```

23.1 SSH with Keyed Access

```
bcome child:server:ssh
```

23.2 From the Console

From the child Inventory namespace:

```
bcome child
ssh server
```

From the server namespace:

```
bcome child
cd server
ssh
```

Hint: Use the 'tree' function or invoke 'ls' on any namespace to see which namespaces are available. See: *Command Menu*.

CHAPTER 24

Interactive Mode

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 (see: *Command Menu*).

Having established an SSH connection to all servers in scope, Interactive mode then provides a secondary interactive pseudo-ssh shell, following which any commands you enter are executed on every server.

> **Warning:** Interactive mode can be dangerous.
>
> You will be executing commands in parallel on every machine in your selection. Be sure you understand what you're running, and with what privileges before using this function.

Interactive mode is useful when managing groups of servers in a real-time scenario for example:

- to apply security patches
- to test en masse for vulnerabilities
- to run system updates
- to compare configurations

Note: Interactive mode will create a connection to every server that is a child or grand-child of the current namespace.

Once connected, connections are cached for speed with reconnections automatic.

For more information on optimising your SSH for interactive mode see *Optimising Ssh*

Interactive mode may be invoked with the `interactive` command on any namespace. See *Executing Commands*.

CHAPTER 25

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).

Run may be invoked with the `run` command on any namespace. See *Executing Commands*.

Note: When in the Console, Bcome caches all SSH connections, with re-connection automatic, resulting in faster execution of repeated ‘run’ commands.

For more information on optimising your SSH connections see *Optimising Ssh*

CHAPTER 26

Tunnel

As Bcome handles your SSH configuration for you via your *Network Configuration*, setting up a tunnel to access a remote service - even one behind any number of proxies - becomes simple.

Let's imagine you want to open up access on local port 9200 to an Elastic Search service running remotely on port 9200, and that your *Namespaces* are setup as follows:

```
.
└── estate
    └── production
        ├── elastic_data_nodes
        │   ├── data_node_1
        │   ├── data_node_2
        │   └── data_node_3
        └── management
            └── jump_host
```

26.1 From the Console

```
bcome production:elastic_data_nodes:data_node_1
the_tunnel = tunnel 9200, 9200
```

The tunnel connection is kept open until the Console session is terminated, or until the tunnel is manually closed, as follows:

```
the_tunnel.close!
```

Note: You may open as many SSH tunnels as you require during a Console session.

26.2 Using Keyed Access

```
bcome production:elastic_data_nodes:data_node_1:tunnel 9200 9200
```

The tunnel connection is kept open until you Control+C to close the session, or until a SIGINT is received by the Bcome process.

26.3 From an orchestration script

Where *@node* is an instance of your server namespace:

```
# Open a tunnel
tunnel = @node.tunnel(9200, 9200)

# Close the tunnel
tunnel.close!
```

Note: You may open as many SSH tunnels as you require from an Orchestration script.

For more information on orchestration scripts see *External Scripts* and *Internal Scripts*.

CHAPTER 27

Pseudo-tty

Bcome's pseudo-tty mode allows you to access a pseudo terminal from any server namespaces.

As Bcome handles your SSH configuration for you via your *Network Configuration*, setting up a pseudo-tty session becomes simple.

This is useful should 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

27.1 How to use pseudo-tty within Bcome

27.1.1 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"
```

27.1.2 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"
```

27.2 Access from the Console

The pseudo_tty function is also accessible directly from the Console.

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

27.3 Incorporating Pseudo-tty sessions as a Registry hook

You 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 Bcome registry hook (see: *Registry Overview*), referencing an internal script (see *Internal Scripts*), within which you would declare the pseudo-tty function as follows:

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

CHAPTER 28

The Metadata Framework

When scripting or otherwise interacting with the Bcome Console, the framework will let you access used-defined metadata.

This is useful when writing orchestration scripts.

As with all other Bcome configuration, Metadata is configured using YAML, and may be encrypted (see *Metadata Encryption*).

28.1 Metadata YAML

To enable metadata you'll need a 'metadata' directory under your 'bcome' configuration directory, as follows:

```
.
└── project
    └── bcome
        └── metadata
```

Any .yml file you place in this directory will be loaded into the framework.

Your Metadata is then defined by declaring attributes onto a namespace breadcrumb.

For example, given a namespace with a breadcrumb of "foo:bar", to assign it Metadata, you would create a .yml file within your 'metadata' directory with the following contents:

```
---
foo:bar:
  key: value
  other_key: other_value
```

28.2 Metadata Inheritance

Metadata is inherited within child namespaces, where it may be overidden.

For example, given two namespace “parent” and “parent:child”, with the following Metadata YAML defined:

```
---
parent:
  key: value
  other_key: other_value

parent:child:
  key: overidden_value
```

The “parent:child” namespace overrides, i.e. re-defines, the ‘key’ attribute, whilst inheriting ‘other_key’. Its Metadata looks like this:

```
---
key: overidden_value
other_key: other_value
```

28.3 Accessing Metadata

See *Metadata Commands* for accessing metadata.

CHAPTER 29

Metadata Encryption

Metadata files (see: *The Metadata Framework*) may be encrypted with a single key.

This allows you to collaborate with others without sharing sensitive data directly (i.e. within your source control system).

The encryption process uses a symmetric block cipher, `AES-256-ECB`

Note: The use of AES-256-ECB will become deprecated in a future release with an intended upgrade to AES-256-CBC. An upgrade path will be provided for already encrypted files.

29.1 Encryption commands

29.1.1 Packing

Encryption is referred to as `packing`.

To pack all your metadata files, invoke the following:

```
bcome pack_metadata
```

You will be prompted for a Metadata key, which will be used to encrypt your data.

Should you now investigate your `metadata` directory, you will see that all your YAML files now have a `.enc` counterpart.

Note: If any metadata YAML file already has a .enc counterpart, you will need to provide the same metadata key used to encrypt that file in order to pack all the others.

Hint: Commit only your .enc files to source control, and create a workflow around Packing & Unpacking.

29.1.2 Unpacking

Decryption is referred to as `unpacking`.

To unpack all your metadata files, invoke the following:

```
bcome unpack_metadata
```

You will be prompted for the same key as was used to Pack your metadata originally.

Warning: Should there be any differences between your .enc metadata files and their .yml counterparts during unpacking, you will prompted for confirmation before proceeding.

The .yml files would otherwise be overwritten with the decrypted contents.

29.1.3 Metadata Diffs

To see the differences between your encrypted metadata and unpacked metadata, use the following command:

```
bcome diff_metadata
```

29.2 Changing your encryption key

To change your encryption key:

- ensure that your unpacked yml metadata contain the correct contents
- remove all *.enc files
- re pack your metadata with a new key, using the `pack_metadata` command.

CHAPTER 30

Metadata Commands

To return a list of all configured metadata:

```
meta # @node.meta
```

To return a Hash of all configured metadata:

```
metadata.all  # or @node.metadata...
```

To fetch a specific metadata value by key

```
metadata.fetch(:key) # or @node.metadata...
```

To fetch a specific metadata value by key, providing a default value should the key not be found:

```
metadata.fetch(:key, { key: "default value }) # or @node.metadata...
```

To return an Array of all configured metadata keys:

```
metadata.keys # or @node.keys
```

CHAPTER 31

Bash scripting

You may execute a local bash script against servers in your collection using the `execute_script` command.

Let's image you have the following *Namespaces* setup:

```
.
└── project
    ├── inventory_one
    │   ├── server_a
    │   ├── server_b
    │   └── server_c
    └── inventory_two
        ├── server_d
        ├── server_e
        └── server_f
```

And the following bash script, saved to your local system at /path/to/script.sh:

```
#!/bin/bash

echo "hello world"

exit 0
```

In order to execute the script against a single server, for example 'server_a' in 'inventory_one', you would invoke the following:

```
bcome inventory_one:server_a:execute_script /path/to/script.sh
```

To execute the script against all servers in 'inventory_two', you would:

```
bcome inventory_two:execute_script /path/to/script.sh
```

Likewise, for all servers in your project:

```
bcome execute_script /path/to/script.sh
```

Note: The examples above illustrate how bash scripts may run using Keyed Access (see: *Navigation*).

Hint: The Console allows for greater flexibility in working with selections of namespaces. See 'Selection Commands' in *Command Menu*.

CHAPTER 32

External Scripts

32.1 Overview

Ruby scripts that run outside the context of your installation are referred to as *external* scripts. These may be run standalone, or integrated into your installation with an external method hook (see: *Registry method types*).

The most basic example of an external script can be seen below:

```
require 'bcome'

# Define an orchestrator
orchestrator = ::Bcome::Orchestrator.instance

# Load in a namespace
namespace = orchestrator.get("some:namespace:breadcrumb")

# Work with your namespace
...
```

All namespaces retrieved by the orchestrator are instances of *@node*.

Note: To return the root namespace using the orchestrator, pass a null breadcrumb i.e. `orchestrator.get()`

See *Executing Commands* for invoking commands and *Command Menu* for a list of commands.

See also *@node methods* for a list of public instance methods.

Hint: Any command available to you in the Console, using Keyed-Access or via the Registry is available to you within your Ruby scripts.

32.2 Some additional useful functions

Prompt for a metadata decryption key:

```
::Bcome::Node::MetaDataLoader.instance.prompt_for_decryption_key
```

Silence command output:

```
::Bcome::Orchestrator.instance.silence_command_output!
```

Initiate an SSH connection to all server instances within a given namespace (rather than lazy-load them):

```
# with a progress bar
::Bcome::Ssh::Connector.connect(namespace, show_progress: true)

# without a progress bar
::Bcome::Ssh::Connector.connect(namespace, show_progress: false)
```

CHAPTER 33

Internal Scripts

Ruby scripts that run inside the context of your installation are referred to as *internal* scripts, and are loaded as extensions to the framework.

Internal scripts are always invoked in the context of a namespace and can be configured to accept arguments (for integration, see *Registry Overview*).

33.1 Getting started

Ensure that you have an 'orchestration' directory within your project's 'bcome' directory, as follows:

```
.
└── project
    └── bcome
        └── orchestration
```

Any Ruby files placed in the 'orchestration' directory will be loaded into the project.

33.2 A basic orchestration script

All internal scripts are ruby classes that inherit from `Bcome::Orchestration::Base` and have a public method named `execute` that takes no parameters.

```
module ::Bcome::Orchestration
  class MyinternalScript < Bcome::Orchestration::Base

    def execute
       # @node = the namespace context
       # @arguments = An argument Hash
    end
```

(continues on next page)

(continued from previous page)

```
  end
end
```

The namespace context in which the internal script was called is available to you in the form of an instance variable named *@node*.

See *Executing Commands* for invoking commands and *Command Menu* for a list of commands.

If arguments have been provided, they are available as a Hash from an instance variable named `@arguments`.

Note: To integrate your internal script into your installation, you must add it to The Registry. See *Registry Overview* and *Registry method types*.

For a guide, see Internal Script Integration Guide.

33.3 Invoking internal scripts from within another

You can trigger an orchestration klass from within another (or in any context within Bcome).

Here's how:

```
script = ::Bcome::Orchestration::MyOtherClass.new(node, arguments)
script.do_execute
```

Where node is an instance of a namespace.

33.4 Traversing namespaces

An internal script is not restricted to the namespace context in which it is called - you may traverse contexts if you know the namespace breadcrumb.

For example, given a namespace referenced by 'my:other:inventory', you may load it as follows:

```
orchestrator = ::Bcome::Orchestrator.instance
node = orchestrator.get("my:other:inventory")
```

Note that the parent root namespace key is implicit.

33.5 Some additional useful functions

Prompt for a metadata decryption key:

```
::Bcome::Node::MetaDataLoader.instance.prompt_for_decryption_key
```

Silence command output:

```
::Bcome::Orchestrator.instance.silence_command_output!
```

Initiate an SSH connection to all server instances within a given namespace (rather than lazy-load them):

```
# with a progress bar
::Bcome::Ssh::Connector.connect(namespace, show_progress: true)

# without a progress bar
::Bcome::Ssh::Connector.connect(namespace, show_progress: false)
```

CHAPTER 34

Registry Overview

The purpose of the Registry is to place custom method hooks directly onto your namespaces. These method hooks are then surfaced within Bcome, and allow you to call custom functionality within the context of the namespaces associated with them.

Each Bcome namespace has its own Registry, the framework giving you a means of adding context-specific method hooks to your application. It is in this way that you are able to build up unique (re-usable) interfaces on top of your namespaces.

Note: The Registry is designed for reusability, and so you may associate the same tasks with different namespaces, DRY'ing up your code.

Have a look also at the *The Metadata Framework*. By associating context-specific data to your namespaces, the same Registry task re-used becomes way more powerful.

Within any namespace in Bcome, invoke the `registry` method to view your available methods.

For configuration, see *Registry Configuration File* and *Registry Configuration Attributes*.

For examples see *Registry method types*.

For further information on Bcome's in-built commands see *Executing Commands* and *Command Menu*.

Hint: Whilst providing convenience accessors for your orchestrative functions, adding Registry methods is the means with which you build up your installation into a self-documenting application tailored to your own requirements.

CHAPTER 35

Registry Configuration File

The Registry is configured with a configuration file named 'registry.yml', that is placed in your 'bcome' configuration directory, as follows:

```
.
└── project
    └── bcome
        └── registry.yml
```

The YAML configuration is a simple Hash structure representing an Array of script declarations, each one keyed on a Regular expression intended to match a specific Bcome namespace breadcrumb pattern.

```
---
 (regular)expression.+:
   - array
   - of
   - available
   - scripts

 (another|pattern)tomatch?:
   - another
   - list
   - of
   - scripts
```

Within Bcome, any namespace with a breadcrumb pattern matching a given Registry declaration's regular expression, will have that script available to it.

Let's imagine you had the following namespace structure:

```
.
└── estate
    ├── aws
    │   ├── dev
    │   │   └── app_servers
```

(continues on next page)

(continued from previous page)

```
    │   └── prod
    │       └── app_servers
    └── gcp
        ├── dev
        │   └── app_servers
        └── prod
            └── app_servers
```

And let's say you need to associate an orchestration script with every 'app_server' inventory.

Your regular expression could look as follows:

```
---
(aws|gcp):(dev|prod):app_servers
...
```

Note: The root namespace name ('estate' in the example above) is always implicit in registry declarations.

For information & examples on configuring scripts, see *Registry method types*. Our Guides site also has example configurations.

CHAPTER 36

Registry Configuration Attributes

Here you'll find the full list of attributes for your registry.yml file (see: *Registry Configuration File*).

attribute key	description	optional
type	the type of Registry declaration: `shortcut`, `internal`, or `external`.	No
description	A description of your command.	No
console_command	The command to invoke within Bcome will invoke your Registry methiod	No
group	A group name. All commands with the same group are grouped together when Registry declarations are listed	No
shortcut_command	The remote command to execute when the script is invoked.	Required for type `shortcut`
run_as_pseudo_tty	true OR false	Optional for type `shortcut` only
orch_klass	Your orchestration class name.	Required for type `internal`
local_command	The system command that is to be executed locally.	Required for type `external`
defaults	A Hash of optional values passed in to your local command. Available as ENV variables to external scripts, and from the @arguments Hash to internal scripts.	Optional for types `external` and `internal`.
restrict_to_node	Set to 'server', 'inventory' (all inventory types), or 'collection'. Registry method will only be made available to namespaces of the declared type.	Yes

Note: See *Registry Configuration File* for an introduction to the registry.yml configuration file.

CHAPTER 37

Registry method types

There are three types of Registry methods:

37.1 Shortcuts

A shortcut references a command that you would otherwise invoke manually.

The example below declares a shortcut to the command `sudo puppetserver ca list --all` made available via a method hook of `list_certs`

```
---
gcp:(dev|prod):xops:puppet:
- type: shortcut
  description: List certificates
  console_command: list_certs
  shortcut_command: "sudo puppetserver ca list --all"
  group: certificates
```

Any namespace with a breadcrumb matching the regular expression `/gcp:(dev|prod):xops:puppet/` would have the list_certs method hook available to it.

37.2 Internal Hooks

An Internal Hook allows for the invoking of Internal ruby scripts.

Below I declare a method hook to a custom orchestration class - `SystemStatus`. The namespace context in which the class is invoked would be made available to the orchestration script instance via the `@node` instance variable.

```
---
gcp:(dev|prod):app_servers:
- type: internal
```

(continues on next page)

(continued from previous page)

```
  description: "Web HTTP status"
  console_command: web_status
  group: status
  orch_klass: SystemStatus
```

Any namespace with a breadcrumb matching the regular expression `/gcp:(dev|prod):app_servers/` would have the 'web_status' method hook available to it.

See *Internal Scripts* for information on how to write your internal scripts.

Note: The full orchestration class namespace is not passed to the orch_klass parameter - only the class name.

37.2.1 Passing arguments

Internal Hook invocations may be passed arguments (see *Registry Configuration Attributes*) when called in Keyed-Access mode (see: *Navigation*).

For example, should you wish to pass an argument 'foo' with a value of 'bar' to the Internal script above for namespace 'gcp:prod:app_servers' you would invoke the following:

```
bcome gcp:prod:app_servers:web_status foo=bar
```

To pass in multiple arguments, you could invoke the following:

```
bcome gcp:prod:app_servers:web_status first=value second=othervalue
```

Within your internal script, your arguments are made available to you within the `@arguments` variable.

37.3 External Hooks

An External Hook allows for the invoking of *External Scripts*

Below I declare a method hook to call a capistrano deployment script.

```
---
"(aws|gcp):(prod|dev):wbzsite(:.+)?":
- type: external
  description: "Deploy web application"
  console_command: deploy
  group: deployment
  local_command: bundle exec cap wbz_frontend deploy build=%build%
  defaults:
    build: "master"
```

When declaring a method hook to an external script, Bcome will append an environment variable named `bcome_context` to the command. This allows you to link your external script to the namespace context in which it was called.

37.3.1 The namespace context

If you invoked the method hook above as follows:

```
bcome gcp:prod:wbzsite:deploy
```

Bcome would execute the following command:

```
bcome_context="gcp:prod:wbzsite" bundle exec cap wbz_frontend deploy build=master
```

Within your external script you would load your namespace context as follows:

```
require 'bcome'

orchestrator = ::Bcome::Orchestrator.instance
namespace = ORCH.get(ENV["bcome_context"])

...
```

37.3.2 Passing arguments

External Hook declarations may be configured to take arguments (see *Registry Configuration Attributes*).

This is achieved using placeholders delineated with `%`. For example should you wish to add 'foo' as an argument attribute to command 'my/command', such that it would be executed as follows -

```
my/command foo=value
```

You would define your 'local_command' attribute within your external hook declaration as follows:

```
---
local_command: my/command foo=%foo%
```

And you would set a default value for foo:

```
---
local_command: my/command foo=%foo%
defaults:
  foo: value
```

Any command argument is made available to your External script as an environnent variable. For example, to load your 'foo' argument within your script:

```
foo = ENV['foo']
```