Contents

Introduction

What is Autostrap?

Autostrap is a framework for deploying, configuring and orchestrating a set of virtual or physical machines that act in concert to provide a service, such as a web shop. It consists of known-good sample configuration for a range of services, sample Heat templates for service clouds (or stacks, as Heat terms them) of one or more machines providing these services, and a set of bootstrapping scripts for setting the machines up for Puppet configuration. It is designed to be easily extended with user or project specific configuration and Puppet code. All components are available under an Apache 2.0 license.

Autostrap has been developed and tested on Ubuntu 14.04. It may work for other Debian flavoured systems, but none have been tested so far.

Environment

Cloudstrap can either run standalone on any Ubuntu 14.04 with an Internet connection or it can be launched through Openstack's Heat orchestration tool. Porting to Cloud environments other than Openstack should not pose too great a challenge, but so far this has not been put to the test.

History

Autostrap started out as an internal Project at Syseleven, where it was used to configure various services running as Heat stacks on Syseleven's Openstack cloud. While it eventually turned out not to be ideal for Syseleven's intended use in managed hosting, it is nonetheless a very useful tool for building infrastructure automatically and reproducibly. Syseleven kindly gave permission to release the code base to the community, the results of which you are looking at now.

How to Read this Document

If you are entirely new to Autostrap, we recommend starting with the Components and Entry Points sections. The former will give you an overview of what components Autostrap consists of, the latter will give you an idea of how and where you can kick off the bootstrapping process. Once you are through with these two sections it's probably best to take some time to read Life of a Stack. It will give you a birds-eye overview of how Autostrap bootstraps a blank VM to a point where it can run puppet.

After that, it's probably best to get your hands dirty and follow the instructions in the Deployment Workflow section to deploy your first Autostrap based service stack.

Finally, you will find detailed reference documentation in the following sections:

Configuration Sources	Discusses all knobs and dials available for configuring Autostrap in detail. Be sure to read this section before building a Autostrap based production setup.
Glossary	A glossary of terms we use in this document.
How do I...	Short howto guides for various common tasks.

Contact and Contributions

For now, use our IRC channel `#autostrap` on FreeNode to get in touch. We do not have a mailing list, yet. For Bug reports/feature requests, please raise an issue on our Github page.

There is no formal contribution process right now. Just submit a pull request on Github. We recommend discussing large and/or breaking changes in the IRC channel first. # Components

In this section we will give a brief rundown of all the components that make up Autostrap. It lists all the git repositories the components of Autostrap reside in and gives a high-level description of these components. All repositories are on Github, with links provided in this section.

A note to users: for setting up your service cloud, you will usually only need a fork of project-config and a checkout of autostrap-utils. The latter is optional but recommended since it provides convenient development tools.

Core Components

bootstrap-scripts	contains the bootstrap scripts that generate a hiera.yaml and get the machine to a point where it can run Puppet.
global-config	contains known-good sample configuration in the shape of [Configuration Topics]{../glossary/topic.md}. Before you implement something from scratch you should browse this repository for an existing solution you might be able to customize to meet your needs.
project-config	is an example repository that demonstrates and documents the structure and semantics of a project specific configuration repository. To use Autostrap, fork this repository and use it as a base for your custom configuration.
autostrap-utils	contains useful utilities for building clouds using Autostrap.

Puppet Modules

puppet-autopuppet	Sets up a Puppet master, Puppet agent or masterless puppet, including Sensu monitoring if desired.
puppet-base	Sets up a sane environment, installs useful packages tweaks sysctls and a range of other things.
puppet-docbuild	Sets up build dependencies and automatically builds this documentation from its mkdocs source.
puppet-openstackfacts	Contains various Facter facts used by Autostrap.
puppet-repodeploy	wraps puppetlabs-vcsrepo so it can be configured through a Hiera hash containing multiple repositories.
puppet-ssh	Configures the SSH daemon and deploys SSH authorized keys.

Entry Points

Depending on the platform Autostrap is deployed on, there are various entry points into the bootstrapping process. We refer to these entry points as *Bootstrapping Stage 0*. The first common point for all platforms is the initialize_instance script. This script we also refer to as *Bootstrapping Stage 1*. It is the starting point for the actual bootstrapping process. Currently, there are the following Stage 0 mechanisms for calling it:

Heat Resource

For Openstack platforms we developed the AS::autostrap Heat resource. It generates a user-data script ready to be passed into OS::Nova::Server instances. This script turns the parameters it receives through Heat into environment variables, clones bootstrap-scripts and launches initialize_instance in this environment. Usually you will only need one instance of this resource per Heat stack, i.e. you can pass the same generated script to all Nova servers in your stack.

autostrap.standalone

For environments without Heat and cloud-init we developed autostrap.standalone. This script provides the same environment as the AS::autostrap Heat resource and invokes initialize_instance as well. It emulates the Heat based parametrization mechanisms detailed in the Life of a Stack section with the following two mechanisms:

- Parameters that are passed into AS::autostrap as Heat properties are retrieved from identically named environment variables.
- cloud-init metadata parameters are passed as `=` delimited key-value arguments to `autostrap.standalone`'s `-m` option.

Example:

```
autostrap.standalone -m topics=puppet-agent \
                     -m nodetype=appserver
```

Life of a Stack

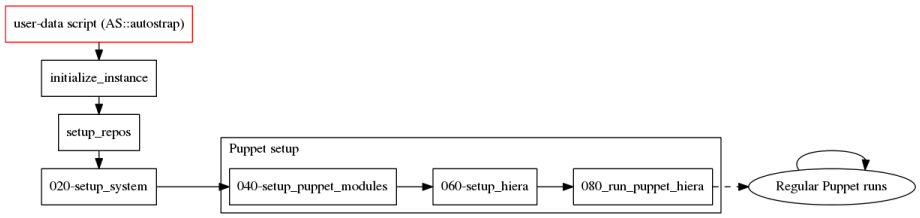
In this section we will describe the life cycle of a Autostrap deployed service stack, from instantiating a heat template to the finished application, in particular this includes:

1. The components a Heat template needs for the instances within to be deployed by Autostrap

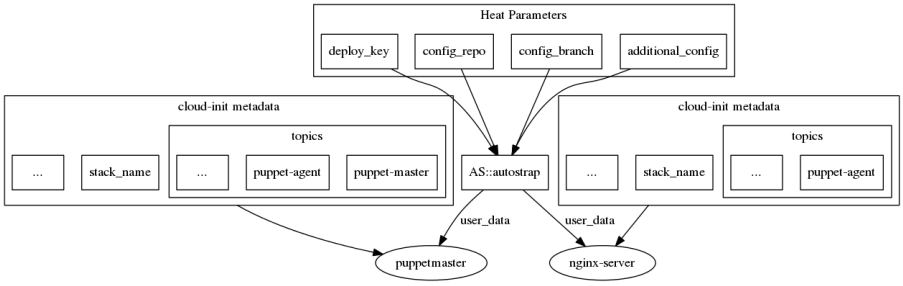
2. The individual stages in an instance's bootstrapping proces.

Read this section to get a high-level overview of what happens during the Autostrap bootstrapping process and how it is kicked off. You can later complement it with the Configuration Sources section for an in-depth discussion of the configuration sources that govern Autostrap's behaviour.

Starting Point: A Heat Template



Once a machine is up, cloud-init will execute the user-data script generated by AS::autostrap. This script will install git, copy the deploy key to /root/.ssh, and clone bootstrap-scripts. Once `bootstrap-scripts` is available, the user-data scripts will execute the `initialize_instance` script in that repository's top-level directory, ushering in the second bootstrapping stage. Deployment of a service stack is kicked off by a Heat template. The schematic below shows the key components of the nginx-master-agent example Heat template and their relation to each other:



There are two kinds of configuration the heat template passes to the two instances (`puppetmaster` and `nginx-server`):

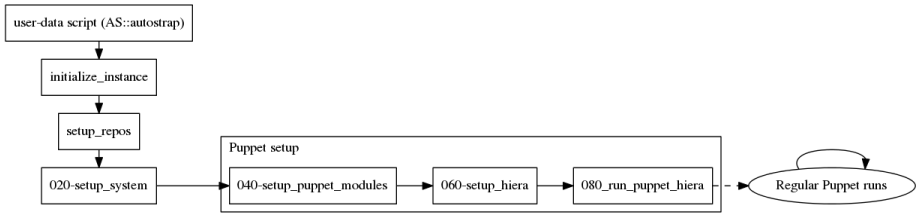
1. A user-data script. This script is generated by the AS::autostrap Heat resource and commonly parametrized through Heat. At a minimum, it needs a `config_repo` parameter pointing at your project-config repository. Commonly there is only one `AS::autostrap` resource that is passed to all servers in the stack.

2. cloud-init metadata keys that pass information for use inside the node and control various aspects of puppet configuration. The most important such parameter is `topics`. This parameter governs the topics from global-config that the second bootstrapping stage's puppet run will deploy on the machine in question. In the example this makes `puppetmaster` a puppet master and agent, and `nginx-server` a puppet agent.

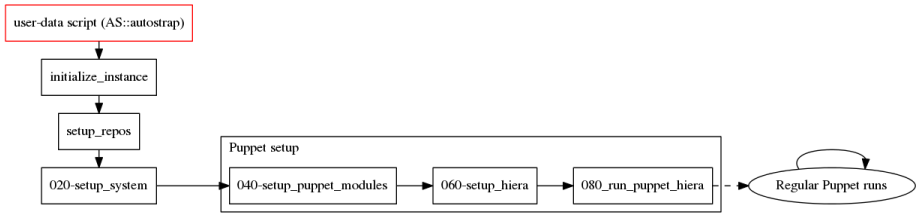
Once the Heat template is finished it can be deployed with a command that might look roughly as follows:

```
heat stack-create -f nginx-master-agent.yaml \
                  -P config_repo=git@gitlab.example.com/my_project_config.git \
                  -P key_name=mykey \
                  -P deploy_key="$(cat ~/.ssh/deploy_key)" \
                  nginx-master-agent
```

Once this command is issued, heat will create the stack's constituent resources. As machines come up, their AS::autostrap generated user-data script will be run by cloud-init. To provide an overview of the bootstrapping process we will use the following schematic throughout the rest of this section, with the current step highlighted:



First Bootstrapping Stage: The AS::autostrap Heat resource



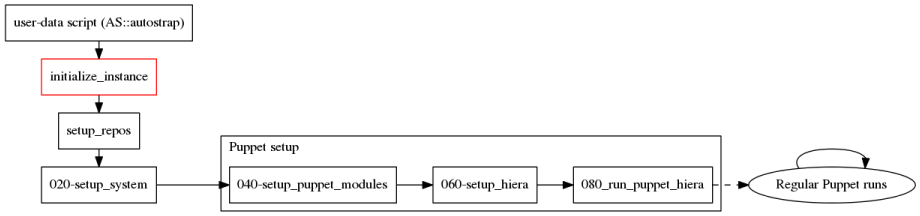
Once a machine is up, cloud-init will execute the user-data script generated by AS::autostrap. This script will install git, copy the deploy key to /root/.ssh,

and clone bootstrap-scripts. Once `bootstrap-scripts` is available, the user-data scripts will execute the `initialize_instance` script in that repository's top-level directory, ushering in the second bootstrapping stage.

Second Bootstraping Stage: The bootstrap-scripts repository

Once the user-data script has run its course, it executes the script `initialize_instance` in the `boostrap-scripts` repository it just cloned.

Starting Point: initialize_instance



`initialize_instance` is primarily a wrapper for starting the various scripts in the second bootstrapping stage. It keeps track of progress (and writes status messages to `/dev/console` for outside visibility) and ensures all second stage output is logged to `/var/log/initialize_instance.log`. Once all second stage scripts have run, `initialize_instance` will report conclusion of the bootstrapping process to `/dev/console`.

All second stage bootstrapping scripts are drawn from two sources:

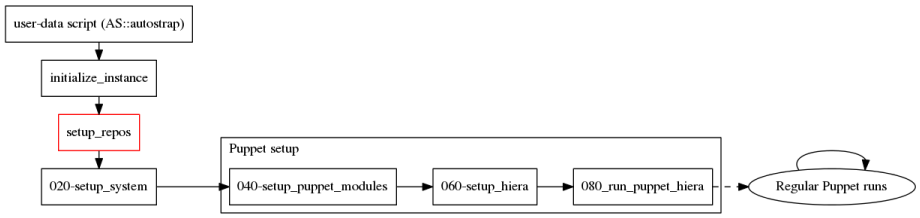
1. The `bootstrap.d/` subdirectory of the global-config repository.
2. The `bootstrap.d/` subdirectory of your project-config repository.

The contents of both these directories are symlinked to the `/opt/scripts/stage/` directory. All files in this directory are executed in shell globbing order, i.e. a script named `000-first` will be executed before 111-last (much like in sysvinit's rc*.d directories). All scripts in `global-config` are prefixed with a zero-padded, three digit multiple of 20, e.g. `000-first`, `020-second`, `040-third`, ...

If you add any scripts to your project-config repository's `bootstrap.d` directory please do not prefix them with multiples of 20 since these are reserved for scripts from global-config. Apart from that anything goes. I.e. you can number them in a way that places them between any two scripts from global-config. If, for instance, you wanted your own script to run between `020-setup_system` and `040-setup_puppet_modules`, you could name it `030-myscript`.

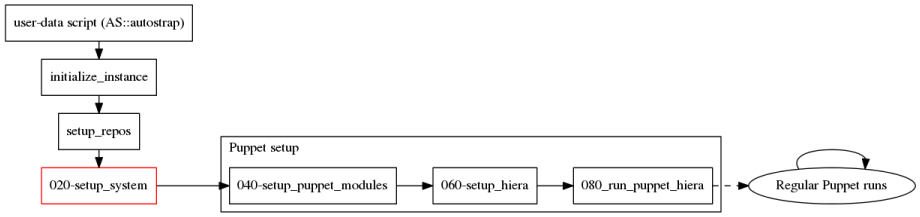
The rest of this section will give a rundown of the bootstrapping scripts pulled in from the global-config repository.

Puppet Setup: Configuration Repository Retrieval



The `setup_repos` script clones the two main configuration repositories that combine into the service stack's configuration (global-config and a project-config repository), as well as the repodeploy puppet module. The URLs and revisions for these repositories are supplied as environment variables passed through from the user-data script.

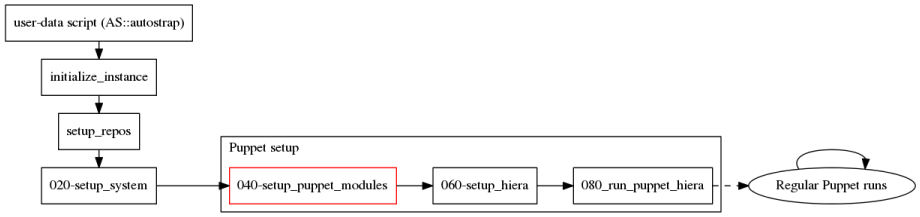
Packages and System Setup



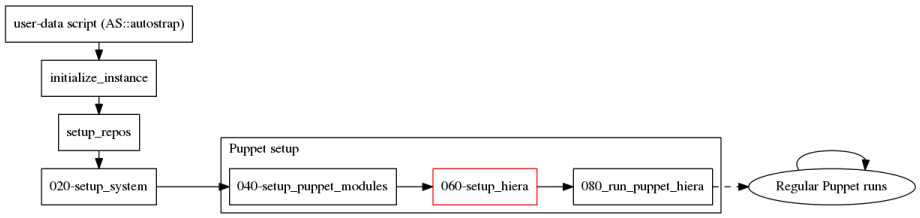
`setup_system` is the first real step in the second bootstrapping stage. It configures `apt` repositories, installs packages required during the second bootstrapping stage, and configures various things in a sensible manner. Once it has finished, Puppet setup begins.

Puppet Setup: Temporary Module Installation

With the preliminaries out of the way, Puppet deployment can now begin in earnest. As a first step, `setup_puppet_modules` performs for puppet what `setup_system` performed for the whole system: it installs a small selection of Puppet modules into `/etc/puppet/modules`. These modules are the bare minimum required for a first run of puppet-repodeploy.



Puppet Setup: Hiera Configuration and Repository Checkouts



`setup_hiera` is the centerpiece of Autostrap's puppet bootstrapping process. It performs two tasks:

1. It generates a `hiera.yaml` configuration file. This file contains a list of all the configuration files that are consulted by puppet running on this machine (both in masterless mode, and by a puppet master if this machine happens to be one).
2. It clones the additional configuration repositories specified in the `additional_config` Heat parameter and adds corresponding entries in hiera.yaml.
3. It runs puppet-repodeploy to retrieve the repositories defined in the `repodeploy::repos` hash. This hash may occur anywhere in the files listed in `hiera.yaml`. Multiple occurences are merged.

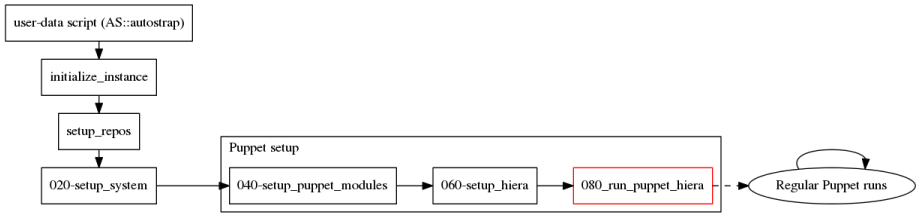
The contents of `hiera.yaml` vary based on the following parameters:

The `cloud-init` metadata entry `topics`	This is a space delimited list that determines the topics to be deployed by (a) `run_puppet_hiera` and (b) subsequent puppet runs, if the `puppet-masterless` topic has been deployed to this machine. All configuration files that make up the topic in question will be entered in `hiera.yaml`.

The contents of `puppet/topics` in the project-config (/glossary/#project-config) repository	This is mainly relevant for a Puppet master. This file contains a list of all the topics this puppet master serves to its agents. `setup_hiera` will add the contents of these topics' `config.d` and `repos.d` subdirectories to `hiera.yaml`. Thus both the configuration relevant to the topics in question and the puppet modules required for their deployment will be available on the puppet master.

Once `hiera.yaml` has been generated and repodeploy has retrieved all repositories, `setup_hiera` will remove `/etc/puppet/modules`, since it has served its purpose: the modules therein were only required to run `puppet-repodeploy`. From this point onward, only puppet modules retrieved by `puppet-repodeploy` will be used. The system is now ready for its first masterless puppet run.

Puppet Setup: First Puppet Run



At the end of the puppet bootstrapping process, `run_puppet_hiera` will run puppet, driven by the `hiera.yaml` just generated. This puppet run will deploy all topics listed in the machine's `topics` metadata entry. This list usually includes either `puppet-masterless` or `puppet-agent`. Hence this first puppet run ensures continuing management of the machine's configuration by regular puppet runs from now on.

Configuration Sources

Machines deployed through Autostrap draw their configuration from various sources. The schematic above shows a big-picture view of these sources and how they interact with each other. In this section we will discuss the four sources for configuration information Autostrap uses:

- Default configuration from global-config
- project specific configuration from a project-config repository
- Additional user specified configuration repositories
- cloud-init user-data and metadata.

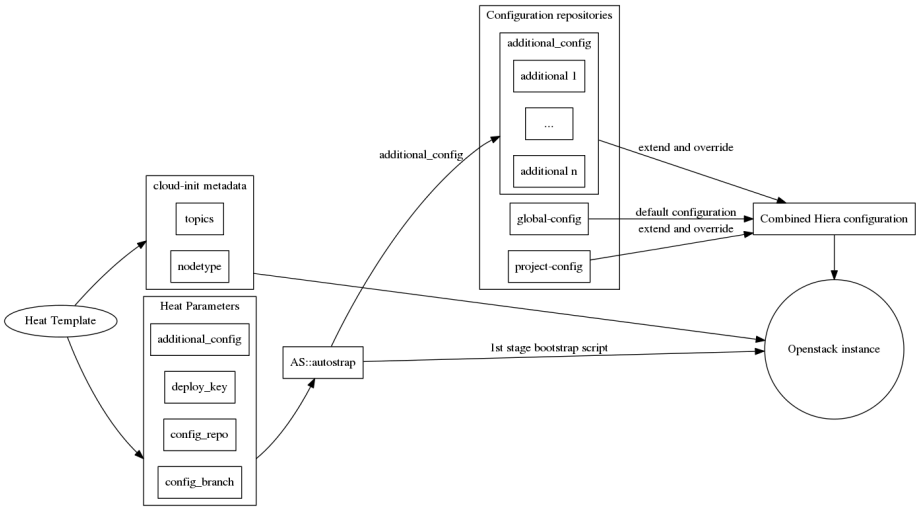


Figure 1: Overview of Autostrap configuration sources

Read this section to get an in-depth tour of the configuration sources at your disposal. If you are itching to try Autostrap you can skip this section. Just create a private fork of project-config and move on to the Deployment Workflow section.

Configuration Sources

global-config: Default Configuration

The repository global-config contains collections of Puppet classes along with matching Hiera configuration to deploy on Openstack instances. We refer to these collections as topics). Topics can be deployed to instances in two ways: directly, using the `topics` metadata parameter, or indirectly by integrating them in a project-config repository.

The `global-config` repository is meant as default configuration, to be overriden and extended by the project-config repository described in the next section.

Project Configuration

The second pilar of configuration is the project-config repository. This is where a Autostrap user will store most of their configuration. Since a fair amount of configuration is provided by global-config, this repository's configuration payload can be fairly small. For instance, the [project configuration content][ex::docserver]

required to set up a web server building and serving this documentation consists of 69 lines at the time of this writing.

Usually a `project-config` repository starts out as a fork of Autostrap's example project-config repository. We recommend you store this fork in a private Git repository which is accessible with an SSH deploy key provided to machines using the `deploy_key` parameter of the AS::autostrap Heat resource. The `project-config` repository's URL and revision are specified through the `config_repo` and `config_branch` parameters to the AS::autostrap Heat resource, respectively.

Additional Configuration

Finally, autostrap contains a mechanism for inserting arbitrary snippets of Hiera configuration. This mechanism is controlled through the AS::autostrap Heat resource's `additional_config` parameter which contains a space delimited list of git repositories. These repositories are cloned during bootstrapping, and entries in hiera.yaml referencing them are generated.

additional_config Parameter Format Entries in the list of repositories are formatted as follows:

```
<repository url>[#<revision>]::[[<path>][:<path> ...]]
```

An entry consists of the following components:

`repository url`	The Repository's URL (mandatory).
`revision`	A revision (branch or commit ID) of the repository to check out (optional).
`path`	A path referencing a YAML file (the .yaml extension may be omitted), relative to the repository's root directory. Each of these paths will result in an entry in hiera.yaml. There may be multiple paths, separated by colons as in a `$PATH` variable. Paths my include shell wildcards (`*`, `?`, etc.) which will be expanded.

additional_config Example This is what the contents of an `additional_config` parameter might look like:

```
'https://example.com/my-additional-config.git::ssh/mykeys.yaml \
git@gitlab.example.com:my-team/my-config.git#devel::config/ssh/keys:apache/*'
```

This will trigger the following actions:

- Clone `https://example.com/my-additional-config.git` and symlink it to `/etc/puppet/hieradata/my-additional-config`.
- Add `my-additional-config/ssh/mykeys` to Hiera's hierarchy.
- Clone `git@gitlab.example.com:my-team/my-config.git`, checkout revision `devel` and symlink it to `/etc/puppet/hieradata/my-team`.
- Add `my-team/ssh/keys`, and all contents of the `apache/` subdirectory to Hiera's hierarchy.

Entry Position in hiera.yaml Hierarchy entries resulting from the `additional_config` parameter will appear before entries from the project-config repository.

Repositories will appear in the order their repository specifications appear in `additional_config`.

Paths associated with a given repository will appear in the order they were given in the repository's `additional_config` entry.

cloud-init: User Data Script and Metadata Parameters

Last, but not least, cloud-init, is the glue that binds the other two configuration sources together. It is used in two ways: to inject a user-data script into an instance and to pass so-called 'metadata', a set of key-value parameters. The user-data script kicks off the bootstrapping process, the metadata parameters influence its behaviour.

cloud-init user-data script The user-data script generated by AS::autostrap is parametrized by its parent heat template and passed to to all servers in a service stack. Through this mechanism the servers receive their deploy keys and their configuration repositories' URLs. This script is usually generated once per service stack and passed to all machines unchanged.

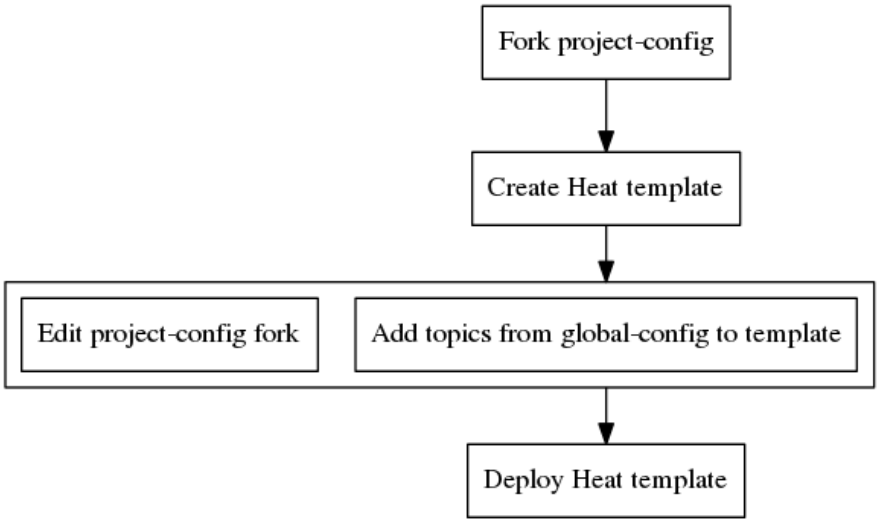
One notable parameter to this script is override_yaml. It contains the contents of `override.yaml`, a file that will appear at the very top of the machine's `hiera.yaml` and override all other configuration. This mechanism is very useful for deploying development stacks based off one or more development branches, or for any other temporary configuration you don't want to commit to your project-config repository.

If you require additional high-level entries, for instance to pull in passwords automatically generated by your bootstraping scripts, you can use the extra_overrides parameter to add arbitrary hierarchy entries between `override.yaml` and additional-config/project-config.

cloud-init metadata parameters

Various `cloud-init` metadata paremeters influence the behaviour of puppet, pass information (such as its floating IP address), or control the topics deployed on a given machine. Metadata parameters can vary on a per-machine basis (e.g. typically only one machine will be assigned the `puppet-master` topic while all others are assigned the `puppet-agent` topic). # Deployment Workflow

In this section we will describe deployment workflows for two kinds of Puppet configuration approaches: masterless Puppet (for single-instance stacks) and a traditional Puppet master/agent setup (for stacks of two or more instances). Both approaches follow the same basic basic steps:



These workflows assume you are deploying to an Openstack cloud, but they can be adapted to autostrap.standalone if you do not have access to an Openstack cloud.

We will start this section off with a check list of common prerequisites you will need for both approaches. Once you have the items on this list covered you can dive right into deploying your first stack using either the master/agent or masterless approach.

Prerequisites

Following the deployment instructions described in this section requires the following:

- A user account on an Openstack cloud

- The Openstack command-line clients
- git
- A remotely accessible (e.g. through SSH or HTTPS) git repository for storing project specific configuration. This repository should start out as a fork of Autostrap's sample project-config repository. Throughout this section we will refer to this repository with the placeholder *my-config* and to its url with the placeholder *my-config-url*.
- An editor with syntax highlighting for YAML (optional, but recommended)
- The syseleven.cloudutils package for smoother handling of heat stacks (optional, but recommended). A simple `pip install syseleven.cloudutils` should take care of this.

Puppet Master/Agent Based Configuration

For larger setups a Puppet master can be automatically deployed, with all nodes (including the Puppet master itself) running Puppet agents. This approach is recommended for any setup beyond two nodes. This approach consists of a masterless first stage that sets up the Puppet master and its agent, with configuration being provided through the Puppet master from there on out.

Step 1: Creating a Heat Template

First of all you will need to describe your projects requirements in terms of a Heat template. As a starting point, the Heat project provides some guidelines on writing heat templates.

Autostrap requires various meta data parameters and its own user-data script, so we recommend you modify one of our example templates in the `heat-templates` directory of project-config to fit your project's requirements.

Adding a AS::autostrap resource

The custom Heat resource `AS::autostrap` generates a user-data script that will kick off the autostrap deployment process. This user-data script is then passed to your instances (`OS::Nova::Server` resources) as user-data property.

Properties The following properties of `AS::autostrap` are most likely to be relevant to deploying a new project (refer to the `AS::autostrap` documentation for the rest):

- `config_repo`: The URL of your project specific configuration repository, i.e. *my-config-url*.

- `config_branch`: An optional branch/revision of your configuration repository to check out. This is mainly useful for development and best sourced from a Heat parameter that defaults to 'master' (thus allowing you to specify experimental/development branches at run-time, while defaulting to a known-good stable branch otherwise).'
- `deploy_key`: A SSH private key with access to all non-public repositories you specified in your repository configuration (i.e. all instances of `repodeploy::repos` occuring in your configuration). We strongly recommend against adding this deploy key to your heat template. Current best practice is to pass it as a parameter (with the `hidden` attribute enabled).

Declaration Example

```
bootstrap:
    type: AS::autostrap::v1
    properties:
      deploy_key: { get_param: deploy_key }
      config_repo: <my-config-url>
      config_branch:
```

Instance Attachment Example

```
my_machine:
  type: OS::Nova::Server
    properties:
      name: my_machine
      user_data: { get_attr: [ bootstrap, script ] }
      user_data_format: RAW
```

Note the "::v1" trailing the resource declaration. Like all of Autostrap's custom resources, this resource is versioned: whenever we introduce breaking API changes we increment the version number. This way you can generally rely on a given version continuing to behave as as expected.

Required Metadata parameters for instances

You will need to pass at least the following heat parameters to instances (`OS::Nova::Server` resources) to be deployed using Autostrap:

- `topics` - The configuration topics to deploy on the instance in question (see Adding Topics to the Puppet Master).

Additionally, it is recommended to set the following metadata parameters:

- `stack_name` - Available to puppet through the fact `openstack_stack_name` (requires the puppet module openstackfacts)
- `floating_ip` - Available to puppet through the fact `openstack_floating_ip` (requires the puppet module openstackfacts)

If you have followed the recommendations above, a regular machine's metadata property might look as follows:

```
metadata:
  stack_name: { get_param: 'OS::stack_name' }
    topics: "base ssh puppet-master-agent"
    stack_name: { get_param: 'OS::stack_name' }
    floating_ip: { get_attr: [ my_port, floating_ip_address ] }
    topics: "base ssh puppet-master-agent"
```

The puppetmaster's metadata property might then look like this:

```
metadata:
  stack_name: { get_param: 'OS::stack_name' }
    topics: "base ssh puppet-master-agent"
    stack_name: { get_param: 'OS::stack_name' }
    floating_ip: { get_attr: [ my_port, floating_ip_address ] }
    topics: "base ssh puppet-master-agent"
```

Step 2: Adding Configuration Topics From global-config

Now you will have to pick configuration topics to deploy from global-config. Each topic consists of a set of puppet classes required to deploy the service or other configuration it deploys, hiera configuration and a list of repositories containing the puppet modules it uses.

At a minimum, you will need the `puppet-master` topic on the designated puppet master and the `puppet-agent` topic on all nodes (including the puppet master). Additionally, `base` (sensible configuration and useful packages) and `ssh` (sensible sshd configuration and rollout of ssh public keys) are highly recommended.

Once you have picked topics, edit your heat template and add them to the `topics` metadata entry of the instances they are to be deployed on. This metadata entry is a simple space separated list. If we assume you picked all topics recommended in the previous paragraph, a machine's meta data would contain the following entry:

```
metadata:
  topics: "base ssh puppet-master-agent"
```

Step 3: Adding Topics to the Puppet Master

The configuration topics you picked in the previous section will be deployed in a masterless fashion upon node initialization. In addition to these you can and should now pick topics from global-config to be available through the puppet master. To this end, simply add their topic names (each on a single line) to the `puppet/topics` file in the *my-config* repository. These topics' `config.d` and `repos.d` subdirectories will be included in the puppet masters Hiera hierarchy, making the topics' configuration available and causing its component puppet modules to be checked out on the puppet master. Declaring these topics' component classes for individual nodes or node types will be up to you (see the next section).

Step 4: Forking and Customizing project-config

Since you probably want to add custom configuration of your own beyond that provided by global-config, you will now need to fork project-config). This fork is the repository we referred to as *my-config* in the Prerequisites section above.

Controlling access to *my-config* *my-config* must be reachable from the Internet, so your Openstack instances can use it to retrieve your projects's configuration. It can be reachable through a public HTTPS URL, but we strongly recommend to make it accessible through SSH with public-key authentication.

If you do make *my-config* non-public you will have to supply a deploy_key property to your `AS::autostrap` resource. This property must contain a SSH private key that can access *my-config*.

Customizing *my-config* Now you can modify *my-config* to your heart's content. The changes and additions will then be deployed on your heat stack's machines. You can put Hiera configuration into the following subdirectories of *my-config*:

- `puppet/hieradata/config.d`: This directory contains hiera configuration relevant to all nodes/node types. It should contain all configuration that is not confined to any specific node or node type. One example for such a configuration value would be the list of SSH authorized keys, configured in the `ssh::keys` hash. File and directory names in this directory are free-form (every file with a .yaml extension will be included), but it is recommended to organize configuration by topic name for easier navigation by developers.
- `puppet/hieradata/repos.d`: This directory contains puppet-repodeploy configuration, i.e. zero or more instances of the `repodeploy::repos` hash.

This hash specifies repositories to be checked out by puppet-repodeploy. File and directory names in this directory are free-form (every file with a .yaml extension will be included), but it is recommended to organize configuration by topic name for easier navigation by developers.

- `puppet/hieradata/nodes.d`: This directory contains node specific configuration. It should only contain configuration that is relevant to specific nodes. All file names in this directory must be the target node's FQDN plus a '.yaml' extension, e.g. `puppetmaster.local.yaml`. All nodes start out with a FQDN of 'hostname'.local. The `classes` array holding the classes to be deployed on a given node should be defined here.

- `puppet/hieradata/nodetypes.d`: This directory contains node type specific configuration. A node's node type (such as `appserver`) is assigned by setting the `nodetype` metadata entry. All file names in this directory must be the target node type's name as given in the `nodetype` metadata entry, plus a '.yaml' extension, e.g. `appserver.yaml`. The `classes` array holding the classes to be deployed on a given node type should be defined here.

Refer to the example project-config repository's README.md file file for more detailed information on these directories and their contents.

Configuring classes to be deployed on nodes and node types All classes to be included in the puppet master's catalog should be declared either on a per-node or a per-nodetype basis, i.e. in `puppet/hieradata/nodes.d` or `puppet/hieradata/nodetypes.d`, respectively. If you wish to deploy one or more of the ready-made topics from global-config, for a given node or node type you will need to do two things:

1. Ensure the topics' configuration and component puppet modules are available on the puppet master (see Adding Topics to the Puppet Master).

2. Add the topics' classes to your nodes' and/or nodetypes' classes arrays.

For the latter step you can use the `merge_classes` script from the autostrap-utils repository. This script will gather multiple topics' classes arrays from global-config and merge them into a classes array ready for inclusion in a configuration file in `nodes.d` or `nodetypes.d`. The following example generates a classes array containing the `nginx-server`, and `puppet-agent` topics:

```
git clone https://github.com/autostrap/global-config.git /tmp/global-config
merge_classes --classes-dir /tmp/global-config/puppet/hieradata/classes.d \
              nginx-server puppet-agent
```

This would yield the following classes array, which also happens to be a valid node/node type configuration:

```
classes:
  - "nginx"
  - "autopuppet::role::agent"
```

To include it in an existing node configuration you would have to remove the `---` in the first line, since leaving it in place would indicate the begin of a new YAML document.

Step 5: Deploying From Your Heat Template

Once configuration is finished and pushed to the *my-config* repository and your heat template contains the neccessary elements you can deploy a heat stack roughly as follows

```
heat stack-create -f mycloud.yaml \
                  -P key_name=<nova key name> \
                  -P deploy_key="$(cat ~/.ssh/deploy_key)" \
                  my-master-agent-stack
```

Substitute the key name you specified when uploading your SSH key to nova for `<nova key name>`. You may also have to supply a `public_net_id` parameter.

Other parameters supplied to `heat stack-create` may vary. This example assumes a largely unmodified copy of the `sensu-master-agent.yaml` example template with just the `config_repo` parameter's default changed to *my-config-url*.

Masterless Puppet Configuration

This approach is meant for small setups that typically consist of only one or two servers. In this case a Puppet master is not really needed. Consequently, all configuration and Puppet modules relevant to the node in question will be locally available on any node. A cronjob will run puppet locally on a regular basis. The diagram below gives an overview of the steps involved in a masterless puppet setup:

Step 1: Creating a Heat Template

First of all you will need to describe your projects requirements in terms of a Heat template. As a starting point, the Heat project provides some guidelines on writing heat templates.

Autostrap requires various meta data parameters and an Autostrap provided user-data script, so we recommend you modify one of our example templates in the heat-templates directory of project-config to fit your project's requirements.

Adding a AS::autostrap resource

The custom Heat resource `AS::autostrap` generates a user-data script that will kick off the autostrap deployment process. This user-data script is then passed to your instances (`OS::Nova::Server` resources) as user-data property.

Properties The following properties of `AS::autostrap` are most likely to be relevant to deploying a new project (refer to the `AS::autostrap` documentation for the rest):

- `config_repo`: The URL of your project specific configuration repository, i.e. *my-config-url*.
- `config_branch`: An optional branch/revision of your configuration repository to check out. This is mainly useful for development and best sourced from a Heat parameter that defaults to 'master' (thus allowing you to specify experimental/development branches at run-time, while defaulting to a known-good stable branch otherwise).'
- `deploy_key`: A SSH private key with access to all non-public repositories you specified in your repository configuration (i.e. all instances of `repodeploy::repos` occuring in your configuration). We strongly recommend against adding this deploy key to your heat template. Current best practice is to pass it as a parameter (with the `hidden` attribute enabled).

Declaration Example

```
bootstrap:
    type: AS::autostrap::v1
    properties:
      deploy_key: { get_param: deploy_key }
      config_repo: <my-config-url>
      config_branch:
```

Instance Attachment Example

```
my_machine:
  type: OS::Nova::Server
    properties:
```

```
name: my_machine
user_data: { get_attr: [ bootstrap, script ] }
user_data_format: RAW
```

Note the "::v1" trailing the resource declaration. Like all of Autostrap's custom resources, this resource is versioned: whenever we introduce breaking API changes we increment the version number. This way you can generally rely on a given version continuing to behave as as expected.

Required Metadata parameters for instances

You will need to pass at least the following heat parameters to instances (`OS::Nova::Server` resources) to be deployed using Autostrap:

- `topics` - The configuration topics to deploy on the instance in question (see Adding Configuration Topics from global-config).

Additionally, it is recommended to set the following metadata parameters:

- `stack_name` - Available to puppet through the fact `openstack_stack_name` (requires the puppet module openstackfacts)
- `floating_ip` - Available to puppet through the fact `openstack_floating_ip` (requires the puppet module openstackfacts)

If you have followed the recommendations above, a machine's metadata property might look as follows:

```
metadata:
  stack_name: { get_param: 'OS::stack_name' }
    topics: "base ssh puppet-masterless"
    stack_name: { get_param: 'OS::stack_name' }
    floating_ip: { get_attr: [ my_port, floating_ip_address ] }
    topics: "base ssh puppet-masterless"
```

Step 2: Adding Configuration Topics From global-config

Now you will have to pick configuration topics to deploy from global-config. Each topic consists of a set of puppet classes required to deploy the service or other configuration it deploys, hiera configuration and a list of repositories containing the puppet modules it uses.

At a minimum, you will need the `puppet-masterless` topic. Additionally, `base` (sensible configuration and useful packages) and `ssh` (sensible sshd configuration and rollout of ssh authorized keys) are highly recommended.

Once you have picked topics, edit your heat template and add them to the `topics` metadata entry of the instances they are to be deployed on. This metadata entry is a simple space separated list. If we assume you picked all topics recommended in the previous paragraph, a machine's meta data would contain the following entry:

```
metadata:
  topics: "base ssh puppet-masterless"
```

Step 3: Forking and Customizing project-config

Since you probably want to add custom configuration of your own beyond that provided by global-config, you will now need to fork project-config. This fork is the repository we referred to as *my-config* in the Prerequisites section above.

Controlling access to *my-config* *my-config* must be reachable from the Internet, so your Openstack instances can use it to retrieve your projects's configuration. It can be reachable through a public HTTPS URL, but we strongly recommend to make it accessible through SSH with public-key authentication.

If you do make *my-config* non-public you will have to supply a deploy_key property to your `As::autostrap` resource. This property must contain a SSH private key that can access *my-config*.

Customizing *my-config* Now you can modify *my-config* to your heart's content. The changes and additions will then be deployed on your heat stack's machines. You can put Hiera configuration into the following subdirectories of *my-config*:

- `puppet/hieradata/config.d`: This directory contains hiera configuration relevant to all nodes/node types. It should contain all configuration that is not confined to any specific node or node type. One example for such a configuration value would be the list of SSH authorized keys, configured in the `ssh::keys` hash. File and directory names in this directory are free-form (every file with a .yaml extension will be included), but it is recommended to organize configuration by topic name for easier navigation by developers.
- `puppet/hieradata/repos.d`: This directory contains puppet-repodeploy configuration, i.e. zero or more instances of the `repodeploy::repos` hash. This hash specifies repositories to be checked out by puppet-repodeploy. File and directory names in this directory are free-form (every file with a .yaml extension will be included), but it is recommended to organize configuration by topic name for easier navigation by developers.

- `puppet/hieradata/nodes.d`: This directory contains node specific configuration. It should only contain configuration that is relevant to specific nodes. All file names in this directory must be the target node's FQDN plus a '.yaml' extension, e.g. `puppetmaster.local.yaml`. All nodes start out with a FQDN of 'hostname'.local.
- `puppet/hieradata/nodetypes.d`: This directory contains node type specific configuration. A node's node type (such as `appserver`) is assigned by setting the `nodetype` metadata entry. All file names in this directory must be the target node type's name as given in the `nodetype` metadata entry, plus a '.yaml' extension, e.g. `appserver.yaml`.

All classes to be included in puppet's catalog should be declared either on a per-node or a per-nodetype basis, unless they are part of a topic from the `topics` metadata entry already, of course. This means adding a `classes` array to a file in `puppet/hieradata/nodes.d` and/or `puppet/hieradata/nodetypes.d`.

Refer to the example project-config repository's README.md file file for more detailed information on these directories and their contents.

Last but not least, you can add last-minute commands to be run at the end of the bootstrapping process to `bootstrap.d/additional`. This is the place for everything that cannot be handled using puppet. This script will run on all nodes using *my-config* as its project configuration repository.

Configuring *my-config* in your Heat template As a final step, you will need to make the Heat stack you are creating aware of your *my-config* repository. To this end, simply pass *my-config-url* as config_repo property to your `AS::autostrap` resource.

Step 5: Deploying From Your Heat Template

Once configuration is finished and pushed to the *my-config* repository and your heat template contains the neccessary elements you can deploy a heat stack roughly as follows:

```
heat stack-create -f mycloud.yaml \
                  -P key_name=<nova key name> \
                  -P deploy_key="$(cat ~/.ssh/deploy_key)" \
                  my-masterless-stack
```

Substitute the key name you specified when uploading your SSH key to nova for `<nova key name>`. You may also have to supply a `public_net_id` parameter.

Other parameters supplied to `heat stack-create` may vary. This example assumes a largely unmodified copy of the `nginx-masterless.yaml` example

template with just the `config_repo` parameter's default changed to *my-config-url*.

How do I...

How do I...

This section contains short HOWTO guides for various common tasks.

...run autostrap.standalone?

First of all you need a blank machine which fullfills the following requirements:

- Ubuntu 14.04
- Internet Access
- Bash, Python, Git
- A deploy key for the private repositories you might use during bootstrapping in `/root/deploy` (optional, but usually neccessary).

We recommend using a throwaway vagrant instance for this purpose.

On your machine, run the following commands:

```
git clone https://github.com/autostrap/bootstrap-scripts.git \
         /tmp/bootstrap-scripts
unset SSH_AUTH_SOCK     # This ensures the deploy key is used
                        # (as opposed to a key an SSH agent might
                        # be forwarding)
cd /tmp/bootstrap-scripts/stage0
deploy_key="$(cat /root/deploy)" ./autostrap.standalone  \
           -m topics="base puppet-masterless" \
           -m nodetype=docbox
```

This example will give you a nginx web server managed through masterless puppet, hosting this documentation. You can vary topics and other metadata parameters and/or environment variables (the Heat properties recognized by `AS::autostrap` are retrieved from identically named environment variables by `autostrap.standalone`) as needed to create other setups.

Note: `autostrap.standalone` will not output anything on the console. To monitor progress tail the following files:

- /var/log/autostrap/stage0.log
- /var/log/initialize_instance.log (only appears once initialize_instance has been launched)

. . . deploy SSH public keys to my Instances?

First of all, you need to enable the ssh topic on the machines you wish to deploy authorized keys on.

Authorized SSH keys can be then be stored anywhere in your Hiera configuration (i.e. in your project-config or an additional_config repository). Variables containing SSH keys must be hashes and have the following format:

```
ssh::keys:
  'mykeyname':
    type: ssh-rsa | ssh-dss
    user: <user>
    key: <raw key>
    ensure: present | absent
```

This hash is keyed by free-form SSH key names. Entries in this hash in turn contain a hash with the following keys/values:

type (required)	The key's type, e.g. `ssh-dss` or `ssh-rsa`.
user (required)	The user whose `~/.ssh/authorized_keys` this key should be added to
key (required)	The raw public key, without the type and comment fields, (e.g. what you'd get from an `awk '{print $2}' ~/.ssh/*.pub`).
ensure (optional)	Whether to ensure the key being present or absent (defaults to present).

There may be multiple variables containing such hashes. You can control which of these are merged to form the final hash of authorized keys in one of two ways:

By default, Autostrap uses the hash `ssh::keys` if the `ssh` topic is enabled for a machine. Typically you would create this as a default hash in global-config and add to it in project-config. To prevent such a hash in global-config (e.g. if you create a project-config for a setup restricted to a smaller circle of keys than the ones found in your global-config repository) from being used you can use one of the following methods:

1. You can set the Hiera variable `ssh::key_sources`. This is an array containing the Hiera variables to retrieve authorized keys from.
2. You can supply the same thing as a space-delimited list of hiera variable names in the metadata parameter `sshkeys`. This is ok for testing, but not recommended in production. If the Hiera variable `ssh::key_sources` is set, it will take precedence over this metadata parameter, i.e. the metadata parameter will be ignored.

If you do not set the list of SSH key sources explicitely, it will default to the following variable:

- `ssh::keys`

Additionally, `ssh::authorized_keys::authorized_keys` will be appended to `ssh::key_sources` if it is defined anywhere in your Hiera configuration. This ensures legacy setups that still store their SSH keys in `ssh::authorized_keys::authorized_keys` will continue to function.

Examples

In this section you will find examples for configuring a custom list of Hiera variables to retrieve SSH authorized keys from, for both approaches outlined above. While you can use either of these approaches, we strongly recommend using the `ssh::key_sources` Hiera variable (it will also take precedence over the metadata approach if both types of configuration are used).

Configuring the SSH key hash names through Hiera To retrieve your SSH key hashes from the variables `ssh::keys::mykeys` and `ssh::keys::myotherkeys` you would add the following to your Hiera configuration:

```
ssh::key_sources:
 - ssh::keys::mykeys
 - ssh::keys::myotherkeys
```

Configuring the SSH key hash names through the `sshkeys` metadata entry To retrieve your SSH key hashes from the variables `ssh::keys::mykeys` and `ssh::keys::myotherkeys` you would add the following entry to the `metadata` key of your `OS::Nova::Server` resources:

```
mynode:
  type: OS::Nova::Server
  properties:
    [...]
    metadata:
      sshkeys: 'ssh::keys::mykeys ssh::keys::myotherkeys'
```

Configuration Topic Reference

This section contains reference documentation on all the topics available from the global-config repository. Each topic's documentation contains an overview of the configuration it ships with and a list of the puppet classes it declares. To enable a given configuration topic on an instance, just add it to this instance's metadata parameter (this parameter is a space delimited list of topics).

base

This topic loads most of Autostrap's puppet-base module (the key handling logic of the ssh topic are part of `puppet-base` but not loaded by the `base` topic. `puppet-base` handles basic system configuration. This includes among other things:

- Setting a root password
- Setting various sysctls to sensible values
- apt(8) configuration
- Installing a few useful packages
- Sane bash, vim and screen settings

Declared Classes

`base::role::common`	Declares most profiles from puppet-base. It does 'not' load `base::role::sshkeys`, for instance (that is declared in the ssh).

nginx-static

This topic sets up a nginx web server that responds to your instance's floating IP address and serves the static files it finds in `/usr/share/nginx/html`. It is

meant for very simple setups, i.e. if you want to do anything beyond serving a bunch of static files through unencrypted HTTP do not use this topic.

Declared Classes

`nginx`	Configures `nginx`.

puppet-agent

This topic configures a puppet agent. It is part of our standard puppet setup process for stacks of two or more instances. If you have a puppet master that is not named `puppetmaster.local`, you will need to set the `puppetmaster` meta data entry to its host name on all instances you deploy this topic to.

Declared Classes

`autopuppet::role::agent`	Configures a puppet agent.

puppet-master

This topic configures a puppet master. It is part of our standard puppet setup process for stacks consisting of two or more instances. Only configure it on the instance you want to be the puppet master.

Declared Classes

`autopuppet::role::puppetmaster`	Configures a puppet master.

puppet-masterless

This topic configures a masterless puppet setup. This is our standard puppet setup for stacks consisting of just one instance.

Declared Classes

`autopuppet::role::masterless`	Configures masterless (local) puppet.

ssh

This topic configures an instance's `ssh` daemon in a secure manner and deploys SSH public keys from several Hiera hashes to the `root` user's `authorized_keys` file (see our key deployment HOWTO for details on configuring authorized keys).

Declared Classes

`ssh`	Our puppet module for ssh configuration.
'base::role::sshkeys'	Deploys authorized keys.

Glossary

In this section you will find explanations of terms we use to refer to various concepts in the Autostrap environment.

project-config

project-config is a Git repository that contains configuration specific to a given project. It complements and overrides Autostrap's default configuration in the global-config repository. For reference on its structure and semantics we provide an example project-config repository. Feel free to fork this repository and use it as a base for your own project-config repository.

topic

A topic is a self-contained unit of default configuration from the global-config repository. It has a topic name (henceforth referred to by the placeholder 'name') and consists of the following directories in `global-config/puppet/hieradata`:

classes.d/`name`	This directory typically contains a file named `classes.yaml` which holds the Hiera array `classes`. The elements of this array are all the puppet classes required to deploy the topic in question.
config.d/`name`	This directory contains all configuration data required by this topic. This configuration can be overriden partially or completely by the configuration in a project-config repository.
repos.d/`name`	This directory contains puppet-repodeploy configuration specifying the repositories to be checked out when deploying the topic in question. Commonly these repositories are puppet modules required for deployment.

Topics can be deployed using masterless Puppet or by by adding it to a Puppet master's configuration. If a topic is deployed to a node in masterless fashion, all the YAML files in the abovementioned directories are added to this node's hiera.yaml. If it is added to the Puppet master's list of topics, only its `config.d` and `repos.d` subdirectories will be added to the Puppet master's hiera.yaml. Class declarations for nodes/node types will have to be configured on a per-node/nodetype basis.