



- [Sign up for free](#)
- [Explore GitHub](#)
- [Features](#)
- [Blog](#)
- [Sign in](#)

- [Star193](#)
- [Fork40](#)

public [leehambley](#) / [capistrano-handbook](#)

- [Code](#)
- [Network](#)
- [Pull Requests 2](#)
- [Issues 7](#)
- [Graphs](#)

- [Tags 0](#)
- [Downloads 0](#)

branch: [master](#)
Switch branches/tags
Filter branches/tags

- [Branches](#)
- [Tags](#)


[jekyll](#)

[master](#)

Nothing to show

- [Files](#)
- [Commits](#)
- [Branches 2](#)

[capistrano-handbook](#) / [index.markdown](#)

 [natedavisolds](#) 3 months ago
[Typo "they syntax" to "the syntax"](#)

[6 contributors](#)



file 425 lines (277 sloc) 20.3 kb

- [Edit](#)
- [Raw](#)
- [Blame](#)
- [History](#)

Capistrano Handbook

Index

- [Simple. The Way it Should Be](#)
- [Building Blocks Variables & Tasks](#)
 - [Tasks](#)
 - [Variables](#)
- [Local Vs. Remote Tasks](#)
- [Rights, Users and Privileges](#)
- [Command Line Options](#)

Anatomy of a Capistrano installation

Typically inside your application you will have a structure not dissimilar to the following:

```
/
|- public/
|- config/
|- application/
```

Regardless of your application's nature you would usually expect to have these directories; whether or not they exist is largely academic.

Capistrano relies on you having a `config` directory into which it will install a `deploy.rb` file that contains your settings.

The `deploy.rb` file is loaded when you call `cap` on the command line; in the event that you aren't in the root of your application (or more accurately that there isn't a `capfile`) in your present working directory, `cap` will search up the directory structure until it finds one, this may include your home directory.

Beware of this when dealing with multiple projects, or nested projects - this feature was intended so that you could run a `deploy`, or open a `cap shell` without moving to the root of your application.

Typically `capifying` an application will create something akin to the following:

```
set :application, "set your application name here"
set :repository, "set your repository location here"

# If you aren't deploying to /u/apps/#{application} on the target
# servers (which is the default), you can specify the actual location
# via the :deploy_to variable:
# set :deploy_to, "/var/www/#{application}"

# If you aren't using Subversion to manage your source code, specify
# your SCM below:
# set :scm, :subversion

role :app, "your app-server here"
role :web, "your web-server here"
role :db, "your db-server here", :primary => true
```

If your application is not separated into `application`, `web` and `database` servers, you can either set them to be the same value; or comment out, or remove the one you do not require.

Typically for a `PHP` application one would expect to comment out the `web` or `app` roles, depending on your requirements. Certain built in tasks expect to run only on one, or the other kind of server.

The `:primary => true` part of the role definitions allows you to have more than one database server, this could easily also be written either of the following two ways:

```
role :db, 'db1.example.com', 'db2.example.com'
-- or --
role :db, 'db1.example.com', 'db2.example.com', :primary => true
```

If you have two servers, and neither is `primary`, or

```
role :db, 'db1.example.com', :primary => true
role :db, 'db2.example.com'
```

If, for example when deploying a Rails application you only wanted `db1` to run migrations, in the first example both might.

Essentially when using the Rails deployment recipes, the `:primary` option defines where database migrations are run.

Similar attributes include `:no_release` often used for the `:web` role by some of the recipes in circulation to decide which servers should not have the code checked out to them.

Attributes like these are arbitrary and you can define some of your own, and use them to filter more precisely where your own tasks run,

You may want to read more about the [role](#) method as it has a few options. There is the alternate `[server]` method which works slightly differently, the examples should demonstrate how-so.

```
role :db, 'www.example.com'
role :app, 'www.example.com'
role :web, 'www.example.com'
```

And the `server` method:

```
server 'www.example.com', :app, :web, :db
```

If you have a lot of multi-function servers, or perhaps just one server running your whole application the `server` method may be a quick and easy way to remove a few LOC and a little confusion from your deployment configuration files.

Other than the shorter syntax, they are functionally equivalent.

Building Blocks Variables & Tasks

Tasks

Tasks are the foundation of a Capistrano setup; collections of tasks are typically called *Recipes*.

Tasks are defined as such, and can be defined anywhere inside your `Capfile` or `deploy.rb`; or indeed any other file you care to load into the `Capfile` at runtime.

```
desc "Search Remote Application Server Libraries"
task :search_libs, :roles => :app do
  run "ls -x1 /usr/lib | grep -i xml"
end
```

Lets break that down a little...

The `desc` method defines the task description, this shows up when using `cap -T` on your application.. these are arbitrary description strings that can be used to help your users or fellow developers.

Tasks without a description will not be listed by a default `cap -T`, but will however be listed with a `cap -Tv`. More command line options for the `cap` script will be discussed later in the handbook.

The `task` method expects a block, that is run when the task is invoked. The task can, typically contain any number of instructions, both to run locally and on your deployment target servers (`app,web,db`).

Namespacing Tasks

It stands to reason that with such a wide scope of available uses, there would be potential for naming clashes, this isn't a problem localized to Capistrano; and elsewhere in the computer sciences world this has been solved with Namespacing; Capistrano is no different, take the following example:

```
desc "Backup Web Server"
task :backup_web_server do
  puts "In Example Backup Web-Server"
end

desc "Backup Database Server"
task :backup_database_server do
  puts "In Example Backup Database-Server"
end
```

Defining a task in this way, and more about how the task blocks are arranged is forthcoming; however imagine we had two tasks `backup` perhaps, that needed to work differently on different roles.. here's how namespaces solve that problem

```
namespace :web_server do
  task :backup do
    puts "In Example Backup Web-Server"
  end
end

namespace :database_server do
  task :backup do
    puts "In Example Backup Database-Server"
  end
end
```

Whilst the tasks in the first example might be listed by `cap -T` as:

```
backup_database_server    Backup Database Server
backup_web_server         Backup Web Server
```

And invoked with either `cap backup_database_server` or `cap backup_web_server`; the second pair of examples would be listed by `cap -T` as

```
database_server:backup    Backup Database Server
web_server:backup         Backup Web Server
```

and similarly invoked with `cap database_server:backup` or `cap web_server:backup`

Namespaces have an implicit default task called if you address the namespace as if it were a task, consider the following example:

```
namespace :backup do

  task :default do
    web
    db
  end

  task :web, :roles => :web do
    puts "Backing Up Web Server"
  end

  task :db, :roles => :db do
    puts "Backing Up DB Server"
  end

end
```

Note: These are nested differently to the two previous examples, as when looked at in these terms it makes a lot more sense to namespace them this way, and simply call the following

- To backup just the web server:

```
$ cap backup:web
```

- To backup just the db server:

```
$ cap backup:db
```

- To back up both in series:

```
$ cap backup
```

It is important to note here that when calling tasks from within tasks, unlike with `rake` where the syntax might be something like `Rake::Tasks['backup:db'].invoke`, with Capistrano you simply name the task as if it were any other ruby method.

When calling tasks cross-namespace, or for readability you can (and often should) prefix the task call with the namespace in which the task resides, for example:

```
namespace :one do
  task :default do
    test
    one.test
    two.test
  end
  task :test do
    puts "Test One Successful!"
  end
end

namespace :two do
  task :test do
    puts "Test Two Successful"
  end
end
```

Calling `cap one` would output:

```
Test One Successful
Test One Successful
Test Two Successful
```

This gets slightly more complicated as the namespace hierarchy becomes more intricate but the same principles always apply.

Namespaces are nestable, an example from one of the core methods is `cap deploy:web:disable` a `disable` task in the `web` namespace which in turn resides in the `deploy` namespace.

There is a `top` namespace for convenience, and it is here that methods defined outside of an explicit `namespace` block reside by default. If you are in a task inside a namespace, and you want to call a task from a *higher* namespace, or outside of them all, prefix it with `top` and you can define the path to the task you require.

Existing Tasks

Unlike `rake` when you define a task that collides with the name of another (within the same namespace) it will overwrite the task rather than adding to it.

Chaining Tasks

As this is considered to be a feature, not a limitation of Capistrano; there is an exceptionally easy way to chain tasks, including but not limited to the `default` task for each namespace, consider the following

```
namespace :deploy do
  # .. this is a default namespace with lots of its own tasks
end

namespace :notifier do
  task :email_the_boss do
    # Implement your plain ruby emailing code here with [`TMail`](http://tmail.rubyforge.org/)
  end
end

after (:deploy, "notifier:email_the_boss")
```

Note the different arguments, essentially it doesn't matter how you send these, strings, symbols or otherwise, they are automatically read through to ascertain what you intended, I could just have easily have written:

```
after ('deploy', "notifier:email_the_boss")
```

The convention here would appear to be, when using a single word namespace, or task name; **pass it as a symbol** otherwise it must be a string, using the colon-separated task notation.

There are both before, and after callbacks that you can use, and there is nothing to stop you interfering with the execution of any method that calls another, take for example that at the time of writing the implementation of `deploy:default` might look something like this:

```
namespace :deploy do
  task :default do
    update
    update_code
    strategy.deploy!
    finalize_update
    symlink
    restart          # <= v2.5.5
  end
end
```

More Info: [Default Execution Path on the Wiki](#)

Here we could inject a task to happen after a symlink, but before a restart by doing something like:

```
after("deploy:symlink") do
  # Some more logic here perhaps
  notifier.email_the_boss
end
```

Which, unless we need the `# Some more logic here perhaps` part could be simplified to:

```
after("deploy:symlink", "notifier:email_the_boss")
```

The first example shows the [shorthand anonymous-task syntax](#).

Calling Tasks

In the examples we have covered how to call tasks on the command line, how to call tasks explicitly in other tasks, and how to leverage the power of callbacks to inject logic into the default deploy strategy. The same techniques of `before()` and `after()` callback usage, and your own tasks and namespaces will become important once you start to get beyond the default deployment steps.

When calling tasks on the command line, most never have to go further than the standard `cap deploy` call; this as you can see from the example above, actually calls a lot of tasks internally, but there is nothing to stop you calling these individually; most rely on other steps, or having queried your choice of `source control` to get the latest revision, but some can be called directly, consider some of the following:

```
$ cap deploy:symlink # re-run the method to symlink releases/<tag> to current/
```

The trivial example above directly calls one task from a namespace from the command line, another more useful example of this might be:

```
namespace :logs do
  task :watch do
    stream("tail -f /u/apps/example.com/log/production.log")
  end
end
```

Which you could then call with:

```
$ cap logs:watch
```

Nothing restricts you calling namespaced tasks directly except their potential data prerequisites.

Another interesting, and often overlooked way of invoking tasks on the command line comes in the form of:

```
$ cap task1 task2 namespace1:task1
```

Which would call, `task1`, `task2`, `namespace1:task1` in order. You can really make use of this; for example you may want to do something like the following to deploy your app, and immediately follow the logs looking for problems.

```
$ cap deploy logs:watch
```

A more interesting application for this technique comes in the form of the [Multi-Stage Extension](#), which qualifies for its own section of the handbook; we'll discuss a simpler implementation briefly here.

The Multi-Stage Extension is designed for deploying the same application to multiple `stages` (development, preview, staging, production, etc) and is usually invoked as such:

```
$ cap production deploy
$ cap production logs:watch
$ cap staging deploy
$ cap staging deploy:rollback logs:watch
```

The Multi-Stage Extension may be implementing something like the following internally:

```
set :application, 'example-website'

task :production do
  set :deploy_to, "/u/apps/#{application}-production/"
  set :deploy_via, :remote_cache
  after('deploy:symlink', 'cache:clear')
end

task :staging do
  set :deploy_to, "/u/apps/#{application}-staging/"
  set :deploy_via, :copy
  after('deploy:symlink', 'cruise_control:build')
end
```

When you call `cap production deploy`, two variables are set to production friendly values, and an callback is added to clear the live cache (however that might need to work for your environment), where when you call `cap staging deploy` those same two variables are given different values, and a different callback is registered to tell your imaginary [Cruise Control](#) server to rebuild and/or test the latest release.

The example above is trivial, but that should explain in a nut shell how the Multi-Stage Extension functions, and how you can implement your own quite easily; The Multi-Stage Extension is still well worth a look, as it is smart about ensuring you don't just run `cap deploy` and get yourself into trouble deploying an application with half of your configuration missing

Transactions

Transactions are a powerful feature of Capistrano that are sadly under-used, *what would happen if your deploy failed?*

Transactions allow us to define what should happen to roll-back a failed task, take a look at the following example:

```
task :deploy do
  transaction do
    update_code
    symlink
  end
end

task :update_code do
  on_rollback { run "rm -rf #{release_path}" }
  source.checkout(release_path)
end

task :symlink do
  on_rollback do
    run <<-EOC
      rm #{current_path};
      ln -s #{previous_release} #{current_path}
    EOC
  end
  run "rm #{current_path}; ln -s #{release_path} #{current_path}"
end
```

Before `deploy:symlink` is run, the only thing required to roll-back the changes made by `deploy:update_code` is to remove the latest release.

In the `deploy:update_code` example, only one step is needed to undo the *damage* done by the failed task, for `deploy:symlink` there is a little more to it, and in this example this is implemented using the `do..end` block syntax also using a [heredoc](#) to pass a multi-line string to the `run()` command, in this instance, as you can see it removes the `current` symlink and replaces it with one to the `previous_release`.

If your roll-back logic was any more complicated than that, you may consider including a rake task with your application with some kind of rollback task that you can invoke to keep the deployment simple.

Variables

Capistrano has its own variable mechanism built in, you will not in the default `deploy.rb` that `capify` generates most of the variable assignment is done in the following manner:

```
set :foo, 'bar'
```

As [set](#) is quite a complex function, we will only brush the surface here.

Here are a few things to note:

```
set :username, 'Capistrano Wizard'

task :say_username do
  puts "Hello #{username}"
end
```

Note that we have a *real* ruby variable to use in our string interpolation, having used the Capistrano specific `set` method to declare, and assign to it.

One of the key benefits to using the `set` method is that it makes the resulting variable available anywhere inside the Capistrano environment, as well as being able to assign complex objects such as Procs to variables for delayed processing.

Set has a partner function [fetch](#) that functions similarly except that it is for retrieving previously `set` variables.

In addition, there is [exists?](#) which can be used to check whether a variable exists at all; this might be used to implement a solution to the *missing stage* problem we left unresolved in the **Tasks** section:


```
before :deploy do
  unless exists?(:deploy_to)
    raise "Please invoke me like `cap stage deploy` where stage is production/staging"
  end
end
```

For convenience Capistrano's internals use a method called `_cset` which is designed to non-destructively set variables, it is implemented using `exists?` and `set`, take a look:

```
def _cset(name, *args, &block)
  unless exists?(name)
    set(name, *args, &block)
  end
end
```

This can be used without you having to redefine it to set a variable, only in the event that it hasn't already been set. If you need to change the value of a variable, please just use `set`.

Part of the argument list to `set` is a `&block`, these can be used to lazy-set a variable, or compute it at runtime... take a look:

```
set :application, 'example-website'
set :deploy_to, { "/u/apps/#{application}-#{stage}" }

task :production do
  set :stage, 'production'
end

task :staging do
  set :stage, 'staging'
end
```

Note that on the second line of the example the `stage` variable doesn't exist, and were Capistrano to evaluate this inline, an exception would be raised.

However, as the `deploy_to` variable isn't used until further through the deployment process, in `deploy:update`, which we know when invoked with `cap production deploy` will run after the `production` task has defined the `stage` variable, the block that is assigned to `:deploy_to` won't be evaluated until then; this is often used by people who wish to have Capistrano ask for their passwords at deploy-time, rather than commit them to the source repository, for example:

```
set(:user) do
  Capistrano::CLI.ui.ask "Give me a ssh user: "
end
```

This prompt won't be displayed until the variable is actually required, which of course depending on the configuration of your callbacks, may be never at all, this is a very valuable feature that can help ensure your low-level staff or colleagues don't have access to sensitive passwords for production environments that you may wish to keep a secret.

Note: The curly-brace, and `do..end` syntaxes are purely a matter of taste and readability, choose whichever suits you better, this is Ruby syntax sugar, and you may use it as you please.

GitHub Links

GitHub

- [About](#)
- [Blog](#)
- [Features](#)
- [Contact & Support](#)
- [Training](#)
- [GitHub Enterprise](#)
- [Site Status](#)

Clients

- [GitHub for Mac](#)
- [GitHub for Windows](#)
- [GitHub for Eclipse](#)
- [GitHub Mobile Apps](#)

Tools

- [Gauges: Web analytics](#)
- [Speaker Deck: Presentations](#)
- [Gist: Code snippets](#)

Extras

- [Job Board](#)
- [GitHub Shop](#)
- [The Octodex](#)

Documentation

- [GitHub Help](#)
- [Developer API](#)
- [GitHub Flavored Markdown](#)
- [GitHub Pages](#)
- [Terms of Service](#)
- [Privacy](#)
- [Security](#)

© 2012 GitHub Inc. All rights reserved.

Markdown Cheat Sheet

Format Text

Headers

```
# This is an <h1> tag
## This is an <h2> tag
##### This is an <h6> tag
```

Text styles

```
*This text will be italic*
_This will also be italic_
**This text will be bold**
__This will also be bold__

*You **can** combine them*
```

Lists

Unordered

```
* Item 1
* Item 2
  * Item 2a
```

- * Item 2b

Ordered

1. Item 1
2. Item 2
3. Item 3
 - * Item 3a
 - * Item 3b

Miscellaneous

Images

! [GitHub Logo] (/images/logo.png)
Format: ! [Alt Text] (url)

Links

<http://github.com> - automatic!
[GitHub] (<http://github.com>)

Blockquotes

As Kanye West said:

```
> We're living the future so  
> the present is our past.
```

Code Examples in Markdown

Syntax highlighting with GFM

```
```javascript  
function fancyAlert(arg) {
 if(arg) {
 $.facebox({div:'#foo'})
 }
}
```
```

Or, indent your code 4 spaces

Here is a Python code example
without syntax highlighting:

```
def foo:  
    if not bar:  
        return true
```

Inline code for comments

I think you should use an
<code>`</code> element here instead.

Something went wrong with that request. Please try again.

Looking for the GitHub logo?

- **GitHub Logo**



[Download](#)

- **The Octocat**



[Download](#)