



Chapters

1. [Introduction](#)
[What is Capistrano?](#)
[What can it do?](#)
[What assumptions does it make?](#)
2. [Quick Start](#)
[Getting started](#)
[Installing Capistrano](#)
[Deployment Recipe](#)
[Setup](#)
[Apache Configuration](#)
[Deploying](#)
[Rolling back a release](#)
3. [A More Complicated Example](#)
[Getting started](#)
[Deployment Recipe](#)
[Spinner](#)
[Spawner](#)
[Reaper](#)
[Deploying](#)
4. [Recipes](#)
[Introduction](#)
[Variables](#)
[Roles](#)
[Tasks](#)
[Extending Tasks](#)
5. [Standard Tasks](#)
[Overview](#)
[cleanup](#)
[cold_deploy](#)
[deploy](#)
[diff_from_last_deploy](#)
[disable_web](#)
[enable_web](#)
[invoke](#)
[migrate](#)
[restart](#)
[rollback](#)
[rollback_code](#)
[setup](#)
[show_tasks](#)
[spinner](#)
[symlink](#)
[update_code](#)
6. [Creating Tasks](#)
[Overview](#)
[run](#)
[sudo](#)
[put](#)
[delete](#)
[render](#)
[transaction](#)
[on_rollback](#)
7. [Extending Capistrano](#)
[Task Libraries](#)
[Writing a Task Library](#)
[Using a Task Library](#)
[Extension Libraries](#)

Options

[exports](#)
[recent changes](#)
[rss 2.0](#) | [atom](#)

Authors

[Login](#) [Signup](#)

4. Recipes

4.1 Introduction

At this point, you’ve seen a few Capistrano recipes. You’ve been exposed to all three of the building blocks of recipes: *variables*, *roles*, and *tasks*. In this chapter, we’ll take a closer look at each of these components and understand better what they can do for us.

4.2 Variables

Capistrano variables are set using the `set` keyword. Once set, you can access them in your recipes by name:

```
Using variables [ruby]  
1 set :application, "flipper"  
2 puts "The application name is #{application}"
```

(Note that because Capistrano recipe files are really just specialized Ruby scripts, you can do most anything in a recipe file that you would be able to do in a full-fledged Ruby script.)

You can set any variables you want. This allows you to create (for instance) configurable tasks that you can then share with others—you just define your tasks to use certain variables, and then others can set those variables in their own scripts. The subversion and darcs scm modules use this approach, allowing you to set (respectively) the `:svn` and `:darcs` variables to define where the executables are on the remote hosts (if they aren’t in the default path).

Capistrano also defines several pre-defined variables internally. Some of the more commonly used of these variables are:

Variable	Default	Description
application	(required)	The name of your application. Used to build other values, like the deployment directory.
repository	(required)	The location of your code’s scm repository.
gateway	nil	The address of the server to use as a gateway. If given, all other connections will be tunneled through this server.
user	(current user)	The name of the user to use when logging into the remote host(s).
password	(prompted)	The password to use for logging into the remote host(s). Probably not a good idea to set this in recipe files, for various reasons.
deploy_to	"/u/apps/#{application}"	The root of the directory tree on the remote host(s) that the application should be deployed to.
version_dir	"releases"	The directory under <code>deploy_to</code> that should contain each deployed revision.
current_dir	"current"	The name to use (relative to <code>deploy_to</code>) for the symlink that points at the current release.
shared_dir	"shared"	The name of the directory under <code>deploy_to</code> that will contain directories and files to be shared between all releases.
revision	(latest revision)	This specifies the revision you want to check out on the remote machines. (Because the definition of a “revision” differs from SCM to SCM, the actual format of this variable is rather free form.)
scm	:subversion	The source control module to use. Currently supported modules are <code>:subversion</code> , <code>:cvs</code> , and <code>:darcs</code> .
svn	(path)	The location on the remote host(s) of the <code>svn</code> executable. This is useful if subversion is installed in a non-standard path on the servers.
checkout	"co"	The subversion operation to use when checking out the code on the remote host. This can be set to “export” if you would rather do an <code>svn export</code> instead of <code>co</code> .
cvs	(path)	The location on the remote host(s) of the <code>cvs</code> executable. This is useful if CVS is installed in a non-standard path on the servers.
darcs	(path)	The location on the remote host(s) of the <code>darcs</code> executable. This is useful if darcs is installed in a non-standard path on the servers.
ssh_options	Hash.new	This is a hash of additional options that you would like passed to the SSH connection routine. This lets you set (among other things) a non-standard port to connect on (<code>ssh_options[:port] = 2345</code>).
use_sudo	true	Whether or not tasks that can use <code>sudo</code> , ought to use <code>sudo</code> . In a shared environment, this is typically not desirable (or possible), and in that case you should set this variable to <code>false</code> , which will cause those tasks to simply try to run the command directly.

One last trick you can use with variables. Sometimes you want a variable to be evaluated lazily, like `deploy_to` is. `deploy_to` is set at the very beginning, by Capistrano, to `"/u/apps/#{application}"`, but at this point the `application` variable has not been set. So what Capistrano does is set the `deploy_to` variable to a `Proc` instance, which gets evaluated the first time `deploy_to` is referenced:

```
Defining a variable to be lazily evaluated [ruby]  
set(:deploy_to) { "/u/apps/#{application}" }
```

Any time you set the value of a variable to be a block (or a `Proc` instance), the first time that variable is accessed the block will be executed, and the return value cached and returned.

4.3 Roles

Roles, as we have seen, allow you to define named subsets of your production servers. You can then define tasks that are only executed on these specific subsets.

To define a new role, you use the `role` keyword, followed by a comma-delimited list of server names that belong in that role. Servers can be put in multiple roles (such as when you have one server that hosts everything).

```
Defining roles [ruby]  
1 role :web, "www.capistrano.com"  
2 role :app, "appl.capistrano.com", "app2.capistrano.com"  
3 role :db, "app2.capistrano.com"  
4 role :spare, "genghis.capistrano.com"
```

You can define as many servers in as many roles as you want. You can even use any name you want for the roles, but Capistrano’s standard roles are written to look for three in particular `web`, `app` and `db`.

If the last parameter to `role` is a Hash, the values will be used to further specialize the servers in that list, creating (in effect) *sub-roles*:

```
Defining roles [ruby]  
1 role :db, "master.capistrano.com", :primary => true  
2 role :db, "slave.capistrano.com"
```

In the above example, there are two servers in the `db` role, so any task associated with the `db` role will be executed on both of them. However, one of the servers (`master.capistrano.com`) is also given the more specific information of `:primary => true` (meaning, in this case, that this server is the *primary* database server). Tasks may then be defined that run only on servers in the `db` role, and with the `:primary => true` setting.

4.4 Tasks

Tasks are like methods. You create them (using the `task` keyword), give them a name and then define what they ought to do. By default, a task is associated with all servers, unless you explicitly specify the subset of servers to be used.

A task may invoke other tasks, simply by naming them. In this sense, a task really is like a method, because it can be invoked anywhere:

```
Defining tasks [ruby]  
1 task :hello_world do  
2   run "echo Hello, $HOSTNAME"  
3 end  
4  
5 task :some_task do  
6   puts "calling hello_world..."  
7   hello_world  
8 end
```

The above example creates two tasks, `hello_world` and `some_task`. Neither task specifies a role, which means that both are potentially associated with all servers. However, let’s look at what this means in practice.

If I execute the `some_task` task, it will print `calling hello_world...` to the terminal, and will then invoke `hello_world`. So far, so good—no servers have been touched, and all activity has been on the local host.

However, when `hello_world` is invoked, it calls `run`. All `run` does is attempt to execute the given command on all associated remote hosts. (We’ll talk more later about the available helper methods, of which `run` is the most commonly used.) This means that as soon as `run` is invoked, Capistrano inspects the current task and determines what roles are active, and then determines which servers those roles map to. If no connection has been made a server yet, the connection is established and cached, and then the command is executed in parallel. *This means that no connections are made to the remote hosts until they are actually needed.*

In the above example, then, no connection is established to any server until `hello_world` is invoked, and then connections are made to all defined servers in all roles. If we only wanted the servers in the `db` and `app` roles to be used for that task, we could specify that:

```
Specifying roles [ruby]  
1 task :hello_world, :roles => [:db, :app] do  
2   run "echo Hello, $HOSTNAME"  
3 end
```

If you only want a single role to be used, you can specify it directly, without putting it in an array (i.e., `:roles => :db`).

As was hinted at earlier in this manual, you can also specify extra information when adding a server to a role:

```
Extra role information [ruby]  
1 role :db, "master.capistrano.com", :primary => true  
2 role :db, "slave.capistrano.com"
```

In the above example, the “master” server has the extra information `:primary => true`, while the “slave” server does not. Both are in the `db` role, but you can define a task that will only execute on the “master” server like this:

```
Using extra information [ruby]  
1 task :hello_world, :roles => :db, :only => { :primary => true } do  
2   run "echo Hello, $HOSTNAME"  
3 end
```

In this case, all servers in the `db` role, with `:primary => true` in their extra information hash, will be targeted for the `hello_action` task.

It should also be mentioned that tasks have complete access to all configuration variables:

```
Accessing configuration variables [ruby]  
1 task :hello_world do  
2   puts "The application is #{application}."  
3   puts "The repository is #{repository}."  
4   puts "Currently using #{scm} as the source control system."  
5   puts "Deploying to #{deploy_to}."  
6   # etc.  
7 end
```

4.5 Extending Tasks

Sometimes, you want to attach some logic to an existing task, either to execute before or after the task itself. For instance, the standard `setup` task builds out the required directories on each of your servers, but what if you have some other specific setup tasks you’d like done at the same time?

Not a problem. Before Capistrano executes a task, it looks for any other task named `before_XYZ` (where `XYZ` is the name of the task to be executed). If it finds such a task, it executes it first. Likewise, when it finishes executing a task successfully, it will look for (and execute) `after_XYZ`.

So, let’s say you want to also create a `shared/cache` directory on each of your servers:

```
Defining an "after" task [ruby]  
1 task :after_setup, :roles => [:web, :app] do  
2   run "mkdir -m 777 #{shared_dir}/cache"  
3 end
```

Notice that you can also specify roles and so forth forth these before and after tasks, so even though `setup` (in this example) executes on all servers, you can have these extra