



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](#)

6. Creating Tasks

6.1 Overview

Creating new tasks is easy. You might even be surprised how easy it is to do fairly complex things. After all, this is all just Ruby code, and anything you can do in Ruby, you can do in a Capistrano task.

There are several methods available to tasks to make your life (and tasks) easier. This chapter will introduce each of them, and show how they can be used.

6.2 run

The `run` helper takes a single string identifying the command to execute. This command can be any valid shell command, or even multiple commands chained together by `&&`. This command (or commands) will be executed on all servers associated with the current task, *in parallel*. If the executed command fails (returns non-zero) on any server, `run` will raise an exception.

Example of using the run helper [ruby]

```
1  run <<-CMD
2    if [[ -d #{release_path}/status.txt ]]; then
3      cat #{release_path}/status.txt
4    fi
5  CMD
```

Additionally, you can pass a block to `run`. The block will be invoked every time the command produces output (`stderr` or `stdout`). The block should accept three parameters: the *channel* (an object representing the underlying SSH channel being used to communicate with the server), the *stream* (a symbol, either `:err` or `:out`), and the *data* itself.

The channel object allows you to send data back to the process, on it's `stdin` stream, by calling `send_data` on the channel. Also, you can access the name of the host that produced the output via `channel[:host]`.

Example of capturing output [ruby]

```
1  run "sudo ls -la" do |channel, stream, data|
2    if data =~ /^Password:/
3      logger.info "#{channel[:host]} asked for password"
4      channel.send_data "mypass\n"
5    end
6  end
```

By default, the `run` command simply echos all output from all hosts to the terminal.

6.3 sudo

The `sudo` command is exactly like the `run` command, except that it executes the command via `sudo`. This assumes that `sudo` is in a standard path on the remote host, *and* that the user you used to log into the server has permission to use `sudo` for the requested operation.

If a password is requested, the password used to log into the server will be used.

sudo example [ruby]

```
sudo "apachectl graceful"
```

Just like `run`, `sudo` can take a block to process output as well.

6.4 put

The `put` helper let's you transfer data from the local host to a file on the remote host. In this case, though, the file is transferred to all associated servers via a single call to `put`. If Net::SFTP is available, it will be used to transfer the files, otherwise a less-robust method is used (pipe to `cat`).

To use `put`, just pass two parameters—a string containing the data to transfer, and the name of the file to receive the data on each remote host. Optionally, you can also specify `:mode => value` to set the mode of the value. (Note, this will overwrite the file on the remote host!)

Using put [ruby]

```
1  put(File.read('templates/database.yml'),
2      "#{release_path}/config/database.yml",
3      :mode => 0444)
```

Also note that unless Net::SFTP is available, `put` cannot be used to (reliably) transfer binary files.

6.5 delete

The `delete` command is just a convenience for executing `rm` via `run`. It just attempts to do an `rm -f` (note the `-f`! Use with caution!) on the remote server(s), for the named file. To do a recursive delete, pass `:recursive => true`:

Demonstrating delete [ruby]

```
delete "#{release_path}/certs", :recursive => true
```

6.6 render

The `render` command is kind of an oddball, since it doesn't change the remote servers at all. It basically just provides an interface for easily rendering ERb templates and returning the result.

So, how does this belong in something like Capistrano?

Consider the [disable_web](#) task. It dynamically generates and stores a `maintenance.html` file on each web server, allowing you to specify a few different components of the presentation (the reason for the downtime, and the estimated end time).

The `render` command makes this easy. You just have a [maintenance.rhtml template](#) that you pass to `render`, along with the variables you want to use in the render, and then pass the result to `put`.

You can use this for all sorts of things—dynamically constructing your `database.yml`, or customizing a script, or whatever you can think of.

If you pass a string to `render`, it is interpreted as the name of a template file to render. The name need not be suffixed with `".rhtml"`—if a file exists with the given name and `".rhtml"` appended to it, that file will be used. The given file must exist relative either to the current directory, or the `capistrano/recipes/templates` directory (for access to standard template files).

Rendering a file [ruby]

```
render "maintenance"
```

The above will render the file `"maintenance.rhtml"` (or `"maintenance"`, if `"maintenance.rhtml"` does not exist) and return the result as a string. You can also specify a hash of variables to use for the render (these will be treated as local variables within the scope of the render):

Rendering a file with variables [ruby]

```
render "maintenance", :deadline => ENV['UNTIL'],
  :reason => ENV['REASON']
```

If you don't want to render a file, but instead have a string containing an ERb template that you want to render, you can do it like this:

Rendering a string [ruby]

```
render :template => "Hello <%= target %>", :target => "world"
```

6.7 transaction

The `transaction` helper lets you execute a series of other tasks with some (limited) ability to roll back their effects if any of them fail. What it really does is execute the attached block, and if an exception is raised it looks to see what tasks have been executed, and then executes the `on_rollback` handler (see below) for each one (if one exists).

This means that the rollback is only as accurate as the `on_rollback` handlers for the associated tasks. And not all tasks specify `on_rollback`.

Using a transaction [ruby]

```
1  task :push_latest do
2    transaction do
3      update_code
4      symlink
5    end
6  end
```

Of the standard tasks, the following define an `on_rollback` handler:

```
disable_web
symlink
update_code
```

6.8 on_rollback

The `on_rollback` helper allows a task to specify a callback to use if that task raises an exception when invoked inside of a transaction (see `transaction`, above). It accepts no parameters, only a block:

Specifying an on_rollback handler [ruby]

```
task :update_code do
  on_rollback { delete release_path, :recursive => true }
  ...
end
```

Note that the `on_rollback` clause is *only* executed when an exception is raised, *when the task is being executed inside the scope of a transaction call*. If the task raises an exception when no transaction is active, the `on_rollback` handler is *not* invoked.