



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](#)

## 5. Standard Tasks

### 5.1 Overview

Capistrano comes with several tasks predefined, almost all of which are targeted specifically at deploying web applications. Some of the tasks are specific to deploying *Rails* applications (keep in mind that Capistrano was originally designed to integrate nicely with Rails).

This chapter will introduce each of the standard tasks, and describe how they can be used, configured, and (where necessary) overridden to achieve your own ends.

Note that at any time you can see what tasks are available—including both your own custom tasks and the standard ones—by running `rake show_deploy_tasks`. Also note that you can look at the definitions for these tasks by finding the `capistrano/recipes/standard.rb` file, located wherever Capistrano was installed.

Finally, as mentioned in [Extending Tasks](#), you can add before and after hooks to any of these tasks, simply by defining a task with the same name and prepending either `before_` or `after_` to it.

### 5.2 cleanup

When you've deployed your application a few times, you'll notice the `releases` directory tends to accumulate a lot of stuff that isn't necessary any more. You'll almost never rollback more than one or two releases if anything goes wrong (but if you do need to, there are more efficient ways of doing it than calling `rollback` over and over).

Thus, this task was introduced in version 0.10.0. The `cleanup` task will delete unused releases, keeping (by default) only the 5 most recent. If you would rather keep more or fewer than 5, you can set the `:keep_releases` variable in your recipe file.

This task runs on all roles. Also, it uses `sudo` to do the delete. There is not currently a way to change this, but if you need to use `run` instead of `sudo`, you can copy the task from the `standard.rb` to your own recipe file and tweak it as necessary.

### 5.3 cold\_deploy

The `cold_deploy` task is used when deploying an application for the first time. It will basically start the application's spinner (via the `spinner` task) and then do a normal deploy. You'll rarely need to use this more than once for an application.

### 5.4 deploy

The `deploy` task is intended to help you push a new release of your software into production. It updates the code on all servers (via the `update_code` and `symlink` tasks), and then restarts the FastCGI listeners on the application servers (via the `restart` task). If you are using a different way of running your applications (like using Apache to manage your FastCGI processes), you may need to override the `restart` task to meet your specific needs.

The `update_code` and `symlink` tasks are executed in a transaction, so if either of them fail your application will be left in its original state.

### 5.5 diff\_from\_last\_deploy

This task simply prints the difference between what was last deployed, and what is currently in your repository. It can be useful for determining what changed since your last deploy.

### 5.6 disable\_web

There are times when you want to temporarily disable web access to your application, such as when you are doing database maintenance, or upgrading your Ruby installation. The `disable_web` task may be used in this instance to put up a static maintenance page that is displayed to visitors, instead of your application.

This task assumes several things:

- You are using Apache to front your applications.
- Your web servers are all in the `:web` role.
- There is a `system` symlink in your application's `public` directory that points to a `#{shared_path}/system` directory.
- You have an rewrite rule set up that redirects all requests to `/system/maintenance.html` if that file exists.

If all three of these conditions hold, all you need to do to disable web access to your application is `rake remote_exec ACTION=disable_web`. If any one of those conditions don't hold for your environment, then you'll need to override the entire `disable_web` task and script it for your specific needs.

Additionally, you can specify the `UNTIL` and `REASON` environment variables, which will be used to tailor the `maintenance.html` file that gets generated. `UNTIL` should be a time (like "10pm UTC") or period ("this evening", or "tomorrow morning")—basically any phrase that can complete the phrase "back by `#{time}`".

The `REASON` environment variable may be used to specify the purpose of the downtime. By default, the word "maintenance" will be used, but any term can be used that will complete the phrase "down for `#{reason}`".

So, you can create a customized maintenance screen by typing:

```
Customized disable_web [shell]

rake remote_exec ACTION=disable_web \
  UNTIL="tomorrow morning" \
  REASON="a vital database ugrade"
```

To help get you started using this task, here's an Apache rewrite condition that looks for and displays the `maintenance.html` page, but only if it exists:

```
Apache rewrite support for disable_web [apache]

1 RewriteCond %{DOCUMENT_ROOT}/system/maintenance.html -f
2 RewriteCond %{SCRIPT_FILENAME} !maintenance.html
3 RewriteRule ^.*$ /system/maintenance.html [L]
```

To re-enable your application, you can use the `enable_web` task.

### 5.7 enable\_web

The `enable_web` task is the reverse of the `disable_web` task, and makes the same assumptions about your environment. All it does is delete the `maintenance.html` file in `#{shared_path}/system`. Assuming your Apache rewrite rules are set up right, deleting that file should be all it takes to unlock your app and let visitors in again.

### 5.8 invoke

For most things, you'll want to create tasks to describe the operations you perform on your servers. However, sometimes there is just a one-off command you want to execute—updating a single file, or dumping the contents of some file. In those instances, you can use the `invoke` task to easily execute some arbitrary command-line on your servers.

To use it, just specify the `COMMAND` environment variable. To restrict the command to a specific set of roles, you can set the `ROLES` environment variable to a comma-delimited list of role names. (By default, the command will be executed on all roles.) Finally, if you want the command to be executed via `sudo`, you can set the `SUDO` environment variable to some non-blank value.

```
Using the invoke task [shell]

rake remote_exec ACTION=invoke \
  COMMAND="svn up /u/apps/flipper/current/app/views" \
  ROLES=app
```

### 5.9 migrate

The `migrate` task exists to help you run ActiveRecord migrations against your production database. It assumes that:

- your database servers are in the `:db` role, and
- your primary database server has `:primary => true` associated with it.

The `migrate` task will only be executed for the `:db` server with `:primary => true`.

By default, all this task does is change to the directory of your current release (as indicated by the `current` symlink), and run `rake RAILS_ENV=production migrate`. You can specify that it should run against the *latest* release (regardless of what the *current* release is) by setting the `migrate_target` variable to `:latest` before invoking this task. Likewise, if you want to specify additional environment variables (beside `RAILS_ENV`) you can set the `migrate_env` variable to the space-delimited list of `name=value` pairs to use.

(Note that for long-running migrations, or those that lock particularly busy tables, you may want to run `disable_web` first to reduce contention for the database.)

### 5.10 restart

The `restart` task is used to restart all FastCGI listeners for your application. It simply calls the `reaper` command, without arguments, which falls back to the default behavior of sending the `USR2` signal to all active processes for your application. (The `spinner/spawner/reaper` setup is described in greater detail in [Chapter 3: A More Complicated Example](#).)

By default, `sudo` is used to invoke the reaper. If your reaper is running as your user and you do not need to use (or have access to) `sudo`, you can set the `:use_sudo` variable to `false`, so that the reaper is invoked via `run` instead.

The `restart` task is only executed on the servers in the `:app` role.

### 5.11 rollback

The `rollback` task will roll your application back to the previously deployed version. It does this by first calling the `update_code` task, and then invoking `restart` to get your FastCGI listeners looking at the right version.

This task can be a lifesaver. Unless you never make any mistakes, someday you're bound to deploy a lemon, and you'll be grateful on that day that you can easily and cleanly rollback to your previous version.

(Note that this only rolls back the code—it does not undo any database migrations that might have been applied by the latest deployment. If you need to rollback database migrations or other wider-ranging environment changes, you can either write your own tasks, or run the `disable_web` task to give you enough time to manually roll the larger changes back. Not a beautiful solution, but as Capistrano matures, so will its ability to cope with these larger issues, out of the box.)

### 5.12 rollback\_code

The `rollback_code` task is primarily used as a single component of the `rollback` task, but it may occasionally be useful on its own. All it does determine what the previous release was (if one exists), update the `current` symlink to point to that, and then delete the latest release. It affects all servers.

### 5.13 setup

The `setup` task only needs to be run once, at the beginning of your application's lifecycle (or any time a new server is added to your production environment). It is non-destructive, though, and may safely be executed against an existing production system.

It runs against all servers, and sets up the expected directory tree. Specifically, it

- Creates the `releases_path` directory and chmods it to `0775`.
- Creates the `shared_path` directory.
- Creates the `shared_path/system` directory and chmods it to `0775`.
- Creates the `shared_path/log` directory and chmods it to `0777`.

You can define additional setup logic by creating an `after_setup` task, which will be called after this task.

### 5.14 show\_tasks

The `show_tasks` task never does any work on any remote servers. All it does is inspect the existing tasks and display them to standard out in alphabetical order, along with their descriptions. This will include both the standard tasks (described here), as well as your own custom tasks.

The default Rails Rakefile makes it easy to execute this task:

```
rake show_deploy_tasks
```

### 5.15 spinner

The `spinner` task may be used to start the `spinner` process for your application (as described in chapter 3). It assumes that you have a file `script/spin` in your application, that describes the process for starting the spinner.

Also, by default the spinner will be started as the `app` user. If you wish to start it as a different user, set the `:spinner_user` variable to something else. (This only works if you are using `sudo` to start the spinner. If you can't use `sudo`, or don't want to use `sudo`, set the `:use_sudo` variable to `false`, and the spinner will always be started as you.)

### 5.16 symlink

The `symlink` task simply attempts to update the `current` symlink to the latest deployed version of the code. You will almost never need to invoke this task directly, but it is used internally by other tasks.

### 5.17 update\_code

The standard `update_code` task will deploy the latest revision of your code to all of your servers. It also does some tweaking and linking to hook up the new release to shared directory. Specifically, this task will:

- Checkout your source code (according to your selected SCM)
- Delete the `log` and `public/system` directories in your new release (if they exist)
- symlink `log` to `#{shared_path}/log`
- symlink `public/system` to `#{shared_path}/system`

Note that because it deletes the `log` and `public/system` directories, you ought not to store anything in those directories that you want put into the production.

This task is frequently extended with after hooks (by creating an `after_update_code` task) to allow you to add application-specific deployment logic. You need to change the permissions on one of your scripts? Or update your `database.yml` or `environment.rb` file dynamically? The `after_update_code` task is where you'll do it.

If `update_code` is run inside of a transaction and it fails for whatever reason (the checkout fails, or whatever), the new release will be deleted from the server, leaving your system in the state it was originally.