



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](#)

3. A More Complicated Example

3.1 Getting started

In the previous chapter, we looked at a simple deployment environment that consisted of a single production box. Although this is a valid environment for small deployments ([Basecamp](#) started out this way, for example), it rapidly becomes untenable as an application grows.

This chapter will revisit the “flipper” application from the previous chapter. Let’s assume a year has passed, and we have much higher usage. The application has definitely outgrown it’s single box. Instead, we’ll do the following:

- Two web servers accessed via load-balancers. The web servers will be running Apache.
- Two application servers accessed via load-balancers from the web servers. The application servers run standalone FastCGI processes.
- Two database servers, one as master, one as slave.

This configuration should allow us to scale nicely to much higher usage. And Capistrano allows us to deploy to this kind of configuration with very little effort.

3.2 Deployment Recipe

The first thing we need to do is revisit our deployment recipe. The roles, in particular need to be completely revisited, and we can also get rid of our custom `restart` task. The complete `deploy.rb` file looks like this:

```
Multi-server deployment recipe [ruby]

1  set :application, "flipper"
2  set :repository,  "http://svn.capistrano.com/flipper/trunk"
3
4  role :web, "www1.capistrano.com", "www2.capistrano.com"
5  role :app, "appl1.capistrano.com", "app2.capistrano.com"
6  role :db,  "db1.capistrano.com", :primary => true
7  role :db,  "db2.capistrano.com"
```

We now have two servers (`www1` and `www2`) in the `web` role, and two servers (`app1` and `app2`) in the `app` role. Fairly self-explanatory.

Looking at the `db` role, though, we have one server (`db1`) with the extra information `:primary => true`. This tells Capistrano that some tasks should be executed only on this server, and not on all `db` servers. (This is useful for things like migrations, where you only want them applied to the primary copy of the data. You could also add `:slave => true` to the `db2` server and then define a backup task that only ran on the slave.)

We can now run the `setup` task again to make sure our directories are all set up on all six machines. Just type:

```
Running setup [shell]

rake remote:exec ACTION=setup
```

3.3 Spinner

Rails comes with three utilities (`spinner`, `spawner`, and `reaper`) for managing your FastCGI processes.

The `spinner` script is located in the `script/process` directory of your application. (If your application doesn’t have this script, you probably just need to update your application to the latest version. Rails 0.13.1 was the last version of Rails *without* the scripts.)

The `spinner` script is intended to be a continually running process that watches the spawned FastCGI processes. When you start the spinner, you also specify a command to invoke that will start your FCGI processes. This command is usually the spawner:

```
Spinner Example [shell]

/u/apps/flipper/current/scripts/process/spinner \
-c '/u/apps/flipper/current/scripts/process/spawner -p 7000 -i 5' \
-d
```

In the above example, the spinner is given the command to execute (the reference to `spawner`, which we’ll describe next), and is told to daemonize (the `-d` switch). By default, the spinner will attempt to execute the given command every 5 seconds. This is an admittedly brute force method of making sure your FastCGI listeners are always up.

Because it is tedious to type the above command frequently, we’ll extract the whole thing into its own script, and put it in `script/spin`.

3.4 Spawner

The `spawner` script is used to spawn multiple FastCGI listeners. You can give it various parameters (try `spawner -h` to see them all), but the notable ones in this context are:

- `-p`: the first port number for the listeners to use
- `-i`: the number of listener instances to start, one per port, starting on the port given by `-p`

Thus, as used above by the `spinner`, each time the spinner executes the `spawner` command (by default, once every 5 seconds), it will try to start 5 FastCGI’s listeners on ports 7000-7004. A listener can’t start if there is already one listening on that port, so only those listeners that have died will actually be respawned.

3.5 Reaper

The `reaper` is the opposite of the `spawner`—it gracefully restarts all running FCGI listeners (sending them `USR2` signals, by default).

The `reaper` also sends (by default) a `USR1` signal to the active `spinner` processes. This causes the spinner to shift into high gear, attempting to restart FastCGI listeners every half second, instead of every 5. Then, when the reaper is done, it drops the spinner back down into low gear. This makes sure that new listeners are started as promptly as possible if the any are killed during the restart.

This means that once the spinner is going, all it takes to restart your FastCGI processes is to invoke the reaper on them. The rest happens automatically.

The `restart` task invokes the `reaper` without arguments by default, so if you want to use a different restart mechanism (i.e., `USR1` to kill the processes instead of `USR2` to restart them) you will need to implement your own `restart` task.

3.6 Deploying

The first deployment is a bit tricky with this setup, because you have to do some bootstrapping. The spinner isn’t running, and you have to *get* it running. But we can’t get it running until we’ve deployed the application...

Not to worry. We’ll just create a couple of custom tasks that will get everything set up for us:

```
Tasks for initial deployment [ruby]

1  desc "Start the spinner daemon"
2  task :spinner, :roles => :app do
3    run "#{current_path}/script/spin"
4  end
5
6  desc "Used only for deploying when the spinner isn't running"
7  task :cold_deploy do
8    transaction do
9      update_code
10     symlink
11   end
12
13   spinner
14 end
```

The first task only applies to the `app` servers, and all it does is start the spinner by invoking our custom `spin` script.

The second task is a more complicated one. It calls the `update_code` and `symlink` tasks in a *transaction*. This means that if either of those tasks fails, they will be rolled back, leaving the system in a consistent state. Once those two tasks finish successfully (executing on all boxes), our new spinner task is invoked (which will only be executed on the `app` servers, remember).

Once that’s all done, you just have to invoke the `cold_deploy` task, and you’re golden!

```
Invoking cold_deploy [shell]

rake remote:exec ACTION=cold_deploy
```

Once you’ve got the spinner running, future deployments can simply use the default `deploy` task:

```
Invoking deploy [shell]

rake deploy
```