Automating Application Deployment with Capistrano (Overview)

Capistrano is a deployment system written on Ruby. Actually you don't have to know ruby if you want to use Capistrano - you'll still be able to implement your basic tasks. But if you want to gain real power and control on your deployment scenarios, some knowledge or ruby will greatly help.

How to install Capistrano

Here is the official Capistrano installation instruction page: https://capistranorb.com/documentation/getting-started/installation/. On Leopard all you need to do is to run the following command with root privileges:

```
gem install -y capistrano
```

On 10.4 you'll have to install ruby and rubygems prior to running this command. The simplest way to do this is to use binary installers.

Must-read article about Capistrano

Basics of using Capistrano are well described on its official site: https://capistranorb.com

Writing simple deployment recipe

Before we write simple deployment recipe, I'd like to highlight 3 things about Capistrano (they are always highlighted in most number of articles about Capistrano, but still...):

- Capistrano reads /etc/capistrano.conf before any (well, almost any) execution. You can use this file to define your roles, for example - in order not to duplicate hostnames across multiple files.
- Capistrano executes all ssh-commands in parallel on several deployment servers. In order to make your deployment recipes simple you better have exactly equal deployment configurations (folder structure, applications versions and so on) on each deployment server.
- Capistrano tasks can be used for anything - not only for deployment. They can run arbitrary ssh commands or, instead, work locally without them.

Ok, back to the deployment recipe. By saying deployment we usually mean "copying already built application or framework to the deployment server". So let's assume that we have an example WO application, called (surprisingly) BugTracker. The build recipes usually have ".cap" extension. Let's create file BugTracker.cap and put it into the BugTracker folder. As we are going to write deployment recipe, we have nothing to do with build process. So let's assume that ant was executed and build product (BugTracker.woa) was placed into the BugTracker/build folder.

So let's start our recipe file:

```
task :deploy, roles => :app do
end
```

This is the empty definition of task "deploy" that will run on application servers (see, roles :app). We need to define the :app role in order to make the recipe usable:

```
role :app, "localhost"
task :deploy, roles => :app do
end
```

Ok - this is not much, but at least something. Capistrano recipes are executed using "cap" command. So now you should be able to execute the following:

```
cap -f BugTracker.cap deploy
```

The output should be:

```
  * executing `deploy'
```

Now let's fill our recipe with some real code. Basically the simplest (not usable in production, but good as example) way of deploying is:

1. Pack the BugTracker.woa into tar.gz archive.
2. Copy the archive to the remote server.
3. Remove previous deployment on the remote server.

4. Unpack the archive.

Let's implement the :deploy task to do this:

1. Pack the BugTracker.woa into tar.gz archive.

This step should be done locally - we don't need to use Capistrano's main feature of executing ssh-commands in parallel on multiple servers.
So the code will be:

```
system "tar -C build -czvf BugTracker.woa.tar.gz BugTracker.woa"
raise "failed to create an archive" unless $?.exitstatus == 0
```

"system" call is one of the standard ruby ways of executing shell commands. The second line checks that exit status of the previous command is 0 (which means success for shell commands). The second line also shows 2 things about ruby:

- The postfix style of conditional operators - it's a kind of syntactic sugar that makes code more readable.
- The $? variable which contains the status of the last executed shell command. You can find out more about this variable from Ruby documentation by executing the following command: "ri Process::Status" ($? contains the object of class Process::Status).

2. Copy the archive to the remote server.

Here's how it can be done:

```
upload "BugTracker.woa.tar.gz", "/tmp/BugTracker.woa.tar.gz"
```

This will copy BugTracker.woa.tar.gz to all servers specified for :app role (because our :deploy task is associated with this role). Capistrano will ask you for username and password to do the copying. It's quite inconvenient to enter username and password each time - so you probably should setup SSH public key authentication.

3. Remove previous deployment on the remote server.

This will look like this:

```
run "rm -rf /Library/WebObjects/Applications/BugTracker.woa"
```

This will remove the /Library/WebObjects/Applications/BugTracker.woa folder on all servers specified for :app role. Notice that we've used "run" instead of "system". There is a great difference between these commands. "system" executes commands locally whereas "run" executes them on all remote servers specified for the current role. Another thing to notice the -f flag used for rm command. Capistrano will throw an exception and exit immediately if one of the commands executed on remote servers fail. Without -f flag rm will fail when there's no "/Library/WebObjects/Applications/BugTracker.woa" folder. This can happen during the first deployment, for example.

4. Unpack the archive.

Nothing new in this code:

```
run "tar -C /Library/WebObjects/Applications -xzvf /tmp/BugTracker.woa.tar.gz"
```

So, right now we have the following deployment script:

```
role :app, "localhost"
task :deploy, roles => :app do
  # creating BugTracker.woa.tar.gz
  system "tar -C build -czvf BugTracker.woa.tar.gz BugTracker.woa"
  raise "failed to create an archive" unless $?.exitstatus == 0

  # copy the BugTracker.woa.tar.gz to remote server
  upload "BugTracker.woa.tar.gz", "/tmp/BugTracker.woa.tar.gz"

  # remove previous /Library/WebObjects/Applications/BugTracker.woa
  run "rm -rf /Library/WebObjects/Applications/BugTracker.woa"

  # unpack /tmp/BugTracker.woa.tar.gz to /Library/WebObjects/Applications"
  run "tar -C /Library/WebObjects/Applications -xzvf /tmp/BugTracker.woa.tar.gz"
end
```

You can change "localhost" to any hostname you want, or even specify several hostnames separated with commas. This script is the simplest one, but it already does plenty of stuff - and can be used in some simple scenarios. Let's complicate things a bit.

Cleanup

Let's write a cleanup task in order not to leave tar.gz archives both on our local machine and on remote servers. The task can look like this:

```
task :cleanup, roles => :app do
  FileUtils.rm_f "BugTracker.woa.tar.gz"

  run "rm -f /tmp/BugTracker.woa.tar.gz"
end
```

The new part here is FileUtils.rm_f call. This is the way to delete files in ruby.
Now we can check that :cleanup task actually works by executing the following command:

```
cap -f BugTracker.cap cleanup
```

It's great to have a cleanup task, but it would be even better if it would run after the deployment. Capistrano has a "hooks" feature that will help us with that:

```
after :deploy, :cleanup
```

Yes, that's all. Now if :deploy tasks finishes successfully :cleanup task will be executed automatically.

Using variables

You can use variable in capistrano scripts. You can set then with the "set" command:

```
set <variable name>, <variable value> - this commands says for itself. Some examples:
```

```
set "var1", "some data"
set :var2, 10
```

Note also that you can use the identifiers starting with ":" as variable names. This is the ruby way of specifying unique identifiers (called symbols in ruby). Using symbols is a bit faster than using strings, besides you can easily see identifiers in your code, as they won't be quoted - and will not look like string literals. Anyway these calls are absolutely equal:

```
set :var1, "some data"
set "var1", "some data"
```

After the variable is set, you can use it in string literals using in the traditional ruby way:

```
run "echo #{var1}"
run "echo #{var2}"
```

So let's generalize our script with some variables usage:

```
role :app, "localhost"

set :app_name, "BugTracker.woa"
set :archive_name, "#{app_name}.tar.gz"
set :tmp_archive_path, "/tmp/#{archive_name}"
set :wo_apps_path, "/Library/WebObjects/Applications"
set :app_path, "#{wo_apps_path}/#{app_name}"

task :deploy, roles => :app do
  # creating BugTracker.woa.tar.gz
  system "tar -C build -czvf #{archive_name} #{app_name}"
  raise "failed to create an archive" unless $?.exitstatus == 0

  # copy the BugTracker.woa.tar.gz to remote server
  upload archive_name, tmp_archive_path

  # remove previous /Library/WebObjects/Applications/BugTracker.woa
  run "rm -rf #{app_path}"

  # unpack /tmp/BugTracker.woa.tar.gz to /Library/WebObjects/Applications"
  run "tar -C #{wo_apps_path} -xzvf #{tmp_archive_path}"
end

task :cleanup, roles => :app do
  FileUtils.rm_f archive_name

  run "rm -f #{tmp_archive_path}"
end
after :deploy, :cleanup
```

Note that in *upload* and *FileUtils.rm_f* calls variable names are used without any additional symbols - that's because they're not the part of any string literal - so they're used as simple ruby variables (well actually things are much more complicated - but at least they look like simple ruby variables).

Moving global definitions to /etc/capistrano.conf

Capistrano processes /etc/capistrano.conf file before processing any recipe. If you use several recipes for multiple projects that are hosted on the same deployment server, you will still have to specify :app role in every recipe. To avoid such duplication you can move the role definition to /etc/capistrano.conf. Also some general variable definitions can be moved there. In our case it's the :wo_apps_path variable.

Conclusion

Actually, with this brief overview of Capistrano features, you'll be able to write quite complicated deployment recipes. But it won't come as a surprise if I say that Capistrano can do a lot more. You can embed capistrano scripts into the ruby code, define multiple deployment configurations in single capistrano file, process output from servers and more and more... I'll write about these topics as soon as possible.