DevOps in Practice

Reliable and automated software delivery

Code Crushing

DANILO SATO

Code Crushing
Books and programming
Rua Vergueiro, 3185 - 8º andar
04101-300 – Vila Mariana – São Paulo – SP – Brasil

Chapter 7

Deployment pipeline

Now that we have adopted the practice of continuous integration, we have a reliable way to generate and validate new versions of the online store on each commit. At the end of a successful build we have a `.war` artifact that becomes a release candidate for production. This is an important step in the journey to implement continuous delivery and increase the deployment frequency. We have all the necessary components to create a click-button deployment process, now we just need to connect the dots.

Currently, the `online_store` Puppet module has a `.war` file with a fixed version that was created when we did our first manual build in chapter 2. To use a more recent `.war` file, we could download a local copy by accessing the job overview page in Jenkins, placing it inside the Puppet module, and reprovisioning the `web` server. But instead of doing this manually, we will learn how to publish artifacts using a package repository that can be accessed directly from our Puppet code during a deploy.

We will also discuss how to integrate the infrastructure code into our automated delivery process and how to model the different steps required to take a code change from commit to production, reliably and effectively.

7.1 Infrastructure affinity: using native packages

Copying files from one place to another is not the most efficient way to deploy. That is why in Java it is common to use `.war` or `.jar` files to group several files in a single package. They are nothing more than a `zip` file – a well known compression format – with some extra metadata that describes the contents of the package. Other languages also have their own formats for packaging and distribution: `.gem` in Ruby, `.dll` in .NET, etc.

System administrators are used to the operating system's native packaging system for packaging, distributing, and installing software. We are already using it extensively in our Puppet code every time we create a resource of the type `Package` or when using the `apt-get` command to install MySQL, Tomcat, Nagios, etc.

This is an example of when developers and system administrators have different opinions and use different tools. DevOps may improve collaboration by simply aligning the tools used during this process. There are several reasons why system administrators prefer to use native packages to install software:

- **Versioning and dependency management**: This reason is questionable because some other formats – such as Rubygems – also have this kind of support. `.jar` and `.war` files also support declaring the package version inside the `META-INF/MANIFEST.MF` file, but this is not mandatory and has no special semantics that tools can take advantage of. Native packages, on the other hand, treat dependencies and different versions as an integral part of the package and know how to resolve them at installation time.

- **Distribution system**: Repositories are the natural way to store and share native packages, and their management and installation tools are able to perform searches and download the required packages at installation time.

- **Installation is transactional and idempotent**: Package management tools support both installing, uninstalling and updating (or upgrading) packages, and these operations are transactional and idempotent. You do not risk installing only half of the package and leaving loose files in the system.

- **Support for configuration files**: Native packages are able to identify configuration files that can be edited after they are installed. The package manager will keep the edited file or will save a copy of the file so you do not lose your changes when you upgrade or remove the package,.

- **Integrity check**: When packages are created, a checksum is calculated based on its contents. After the package has been downloaded for installation, the package manager will recalculate this checksum and compare it with what was published in the repository to ensure that the package has not been tempered or corrupted during the download process.

- **Signature check**: Similarly, packages are cryptographically signed when published in the repository. During the install process, the package manager can check this signature to ensure that the package is actually coming from the desired repository.

- **Audit and traceability**: Package managers allow you to discover which package installed which file on the system. Moreover, you will discover where a certain package came from and who was responsible for creating it.

- **Affinity with infrastructure tools**: Finally, most infrastructure automation tools – such as Puppet, Chef, Ansible, Salt, etc. – are able to deal with native packages, package managers and their repositories.

For these reasons we will learn how to create a native package for our application. Furthermore, we will also create and configure a package repository to publish and distribute these new packages generated at the end of each successful build.

Provisioning the package repository

Our servers are virtual machines running Linux as their operating system. Specifically, we are using Ubuntu, a Linux distribution based on Debian. In this platform, native packages are known as `.deb` packages and `APT` is the standard package manager tool.

A package repository is nothing more than a well defined directory and file structure. The repository can be exposed in various ways, such as: HTTP, FTP, a local file system, or even a CD-ROM, which used to be the most common form of distributing Linux.

We will use **Reprepro** (http://mirrorer.alioth.debian.org/) to create and manage our package repository. We will reuse the `ci` server to distribute the packages because we will be able to use the same tool to manage the repository and to publish new packages at the end of each build. For this we will create a new class in our `online_store` module called `online_store::repo` within a new `modules/online_store/manifests/repo.pp` file with the following initial content:

```
class online_store::repo($basedir, $name) {
  package { 'reprepro':
    ensure => 'installed',
  }
}
```

This will install the Reprepro package. This class receives two parameters: `$basedir` will be the full path where the local repository directory will be created and `$name` is the name of the repository. We also need to include this class in the `ci` server and we will do this by changing the `online_store::ci` class in the `modules/online_store/manifests/ci.pp` file:

```
class online_store::ci {
  ...
  $archive_artifacts = 'combined/target/*.war'
  $repo_dir = '/var/lib/apt/repo'
  $repo_name = 'devopspkgs'

  file { $job_structure: ... }
  file { "${job_structure[1]}/config.xml": ... }

  class { 'online_store::repo':
    basedir => $repo_dir,
    name    => $repo_name,
  }
}
```

We added two new variables to represent the root directory and the name of the Repository, and we created a new `Class['online_store::repo']` resource that uses these variables as class parameters.

In Unbutu, each version of the operating sytem has a nickname, also known as **distribution**. In our case, we are using Ubuntu 12.04, also known as `precise`.

Debian and Ubuntu repositories are also divided into **components** that represent different levels of support: `main` contains software that is officially supported and free; `restricted` has software that is supported but with a more restrictive license; `universe` contains packages maintained by the community in general; `multiverse` contains software that is not free.

In our case, since we are distributing only a single package, all of these classifications are not as important, so we chose to distribute our package as a `main` component.

For Reprepro to create the initial directories and files structure of the repository, we must create a configuration file called `distributions` within a `conf` directory in the root of the repository. For that, we create a new ERB template file under `modules/online_store/templates/distributions.erb` with the following content:

```
Codename: <%= name %>
Architectures: i386
Components: main
SignWith: default
```