DevOps in Practice

Reliable and automated software delivery

Casa do Código

DANILO SATO

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

Dedication

“To my dad, who introduced me to the world of computers and is my life-long role model."

Preface

Jez Humble

Shortly after I graduated from university in 1999, I got a job at a start-up in London. My boss, Jonny LeRoy, taught me the practice of continuous deployment: When we were finished with a new feature, we would do a quick manual smoke test on our workstation and then ftp the relevant ASP scripts directly onto the production server – not a practice I would recommend today, but it did have the advantage of enabling us to get new ideas to our users quickly.

In 2004 I joined ThoughtWorks where my job was to help enterprises deliver software, and I was appalled to discover that lead times of months or even years were common. Fortunately, I was lucky enough to work with a number of smart people in our industry who were exploring how to improve these outcomes while also increasing quality and improving our ability to serve our users. The practices we came up with also made life better for the people we were working with (for example, no more deployments outside of business hours) – an important indication that we were doing something right. In 2010, Dave Farley and I published " Continuous Delivery," in which we describe the principles and practices that make it possible to deliver small, incremental changes quickly, cheaply, and at low risk.

However, our book omits the nuts and bolts of how one actually gets started creating a deployment pipeline, put in place monitoring and infrastructure as code, and the other important, practical steps needed to implement continuous delivery. Thus I am delighted that Danilo has written the book that you have in front of you, which I think is an important and valuable contribution to the field. Danilo has been deeply involved in helping organizations implement the practices of continuous delivery for several years and has deep experience, and I am sure you will find his book practical and

informative.

I wish you all the best with your journey.

About the book

Delivering software in production is a process that has become increasingly difficult in IT department of various companies. Long testing cycles and a division between development and operations teams are some of the factors that contribute to this problem. Even Agile teams that produce releasable software at the end of each iteration are unable to get to production when they encounter these barriers.

DevOps is a cultural and professional movement that is trying to break down those barriers. Focusing on automation, collaboration, tools and knowledge sharing, DevOps is showing that developers and system engineers have much to learn from each other.

In this book, we show how to implement DevOps and Continuous Delivery practices to increase the deployment frequency in your company, while also increasing the production system's stability and reliability. You will learn how to automate the build and deployment process for a web application, how to automate infrastructure configuration, how to monitor the production system, as well as how to evolve the architecture and migrate it to the cloud, in addition to learning several tools that you can apply at work.

Acknowledgements

To my father, Marcos, for always being an example to follow and for going beyond by trying to follow the code examples without any knowledge of the subject. To my mother, Solange, and my sister, Carolina, for encouraging me and correcting several typos and grammar mistakes on preliminary versions of the book.

To my partner and best friend, Jenny, for her care and support during the many hours I spent working on the book.

To my editor, Paulo Silveira, for giving me the chance, and knowing how to encourage me at the right time in order for the book to become a reality. To my reviewer and friend, Vivian Matsui, for correcting all my grammar mistakes.

To my technical reviewers: Hugo Corbucci, Daniel Cordeiro and Carlos Vilella. Thanks for helping me find better ways to explain difficult concepts, for reviewing terminology, for questioning my technical decisions and helping me improve the overall contents of the book.

To my colleagues Prasanna Pendse, Emily Rosengren, Eldon Almeida and other members of the "Blogger's Bloc" group at ThoughtWorks, for encouraging me to write more, as well as as for providing feedback on the early chapters.

To my many other colleagues at ThoughtWorks, especially Rolf Russell, Brandon Byars and Jez Humble, who heard my thoughts on the book and helped me choose the best way to approach each subject, chapter by chapter.

Finally, to everyone who contributed directly or indirectly in writing this book.

Thank you so much!

About the author

Danilo Sato started to program as a child at a time when many people still did not have home computers. In 2000, he entered the bachelor's program in Computer Science at the University of São Paulo [USP], beginning his career as a Linux Network Administrator at IME-USP for 2 years. While at university he began working as a Java / J2EE developer and had his first contact with Agile in an Extreme Programming (XP) class.

He started his Masters at USP soon after graduation, and supervised by Professor Alfredo Goldman, he presented his dissertation in August 2007 about "Effective Use of Metrics in Agile Software Development" [15].

During his career, Danilo has been a consultant, developer, systems administrator, analyst, systems engineer, teacher, architect and coach, becoming a Lead Consultant at ThoughtWorks in 2008, where he worked in Ruby, Python and Java projects in Brazil, USA and United Kingdom. Currently, Danilo has been helping customers adopt DevOps and Continuous Delivery practices to reduce the time between having an idea, implementing it and running it in through production.

Danilo also has experience as a speaker at international conferences, presenting talks and workshops in: 2007/2009/2010 XP, Agile 2008/2009, Ágiles 2008, Java Connection 2007, Falando em Agile 2008, Rio On Rails 2007, PyCon Brazil 2007, SP RejectConf 2007, Rails Summit Latin America 2008, Agile Brazil 2011/2012/2013, QCon SP 2011/2013, QCon Rio 2014, RubyConf Brazil 2012/2013. He was also the founder of the Coding Dojo @ São Paulo and an organizer for Agile Brazil 2010, 2011 and 2012.

Contents

Versão: 19.3.17

Chapter 1

Introduction

With the advancement of technology, software has become an essential part of everyday life for most companies. When planning a family vacation — scheduling hotel rooms, buying airplane tickets, shopping, sending an SMS or sharing photos of a trip — people interact with a variety of software systems. When these systems are down, it creates a problem not only for the company that is losing business, but also for the users who fail to accomplish their goals. For this reason, it is important to invest in quality software and stability from the moment that the first line of code is written until the moment it starts running.

1.1 Traditional approach

Software development methodologies have evolved, but the process of transforming ideas into code still involves several activities such as requirements

gathering, design, architecture, implementation and testing. Agile software development methods have emerged in the late 90s, proposing a new approach to organize such activities. Rather than performing them in distinct phases - the process known as waterfall - they happen at the same time, in short iterations. At the end of each iteration, the software becomes more and more useful, with new features and less bugs, and the team decides with the customer what should be the next slice that is developed.

As soon as the customer decides that the software is ready to go live and the code is released to production, the real users can start using the system. At this time, several other concerns become relevant: support, monitoring, security, availability, performance, usability, among others. When the software is in production, the priority is to keep it running stably. In cases of failure or disaster, the team needs to be prepared to react quickly to solve the problem.

Due to the nature of these activities, many IT departments have a clear separation of responsibilities between the **development team** and the **operations team.** The development team is responsible for creating new products and applications, adding features or fixing bugs, while the operations team is responsible for taking care of these products and applications in production. The development team is encouraged to introduce changes, while the operations team is responsible for keeping things stable.

At first glance, this division of responsibilities seems to make sense. Each team has different goals and different ways of working. While the development team works in iterations, the operations team needs to react instantly when something goes wrong. Furthermore, the tools and knowledge necessary to work in these teams are different. The development team evolves the system by introducing changes. On the other hand, the operations team avoids changes because they bring a certain risk to the stability of the system. This creates a conflict of interest between these two teams.

Once the conflict exists, the most common way to manage this relationship is by creating processes that define the method of working as well as the responsibilities of each team. From time to time, the development team packages the software that needs to go to production, writes some documentation explaining how to configure the system, how to install it in production, and then transfers the responsibility to the operations team. It is common to use

ticket tracking systems to manage the communication between the teams and defining service-level agreements (SLAs) to ensure that the tickets are processed and closed in a timely fashion. This hand-off often creates a bottleneck in the process of taking code from development and testing to production. It is common to call this process a **deployment** to production, or simply a production **deploy**.

Over time, the process tends to become more and more bureaucratic, decreasing the frequency of deploys. With that, the number of changes introduced in each deploy tends to accumulate, also increasing the risk of each deploy and creating the vicious cycle shown in figure 1.1.

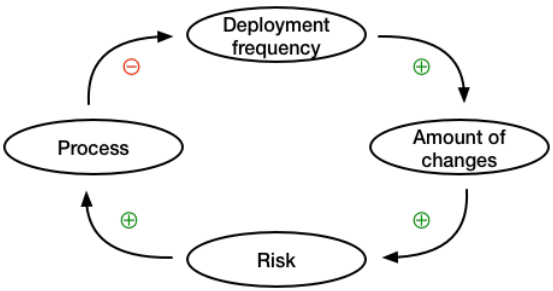

Fig. 1.1: Vicious cycle between development and operations

This vicious cycle not only decreases the ability of the company to respond quickly to changes in business, but also impacts earlier stages of the development process. The separation between development and operation teams, the hand-off of code between them and the ceremony involved in the deployment process, end up creating a problem known as **"the Last Mile"** [17].

The last mile refers to the final stage of the development process that takes place after the software meet all its functional requirements but before being deployed into production. It involves several activities to verify whether the software that will be delivered is stable or not, such as: integration testing, system testing, performance testing, security testing, user acceptance testing (UAT), usability testing, smoke testing, data migration, etc.

It is easy to ignore the last mile when the team is producing and showcasing new features every one or two weeks. However, there are few teams

that are actually deploying to production at the end of each iteration. From the business point of view, the company will only have a return of investment when the software is actually running in production. The last mile problem is only visible when taking a holistic view of the process. To solve it we must look past the barriers between the different teams involved (the business team, the development team or the operations team).

1.2 An alternative approach: DevOps and Continuous Delivery

Many successful internet businesses — such as Google, Amazon, Netflix, Flickr, Facebook and GitHub — realized that technology can be used in their favor and that delaying a production deploy means delaying their ability to compete and adapt to changes in the market. It is common for them to perform dozens or even hundreds of deploys per day!

The line of thinking that attempts to decrease the time between the creation of an idea and its implementation in production is also known as **"Continuous Delivery"** [11], and is revolutionizing the process of developing and delivering software.

When the deployment process ceases to be a ceremony and starts becoming commonplace, the vicious cycle of figure 1.1 gets completely reversed. Increasing deployment frequency causes the amount of change in each deploy to decrease, also reducing the risk associated with that deploy. This benefit is not something intuitive, but when something goes wrong it is much easier to find out what happened because the amount of changes that may have caused the problem is smaller.

However, reducing the risk does not imply a complete removal of the processes between development and operation teams. The key factor that allows the reversal of the cycle is process automation, as shown in figure 1.2. Automating the deployment process allows it to run reliably at any time, removing the risk of problems caused by human error.

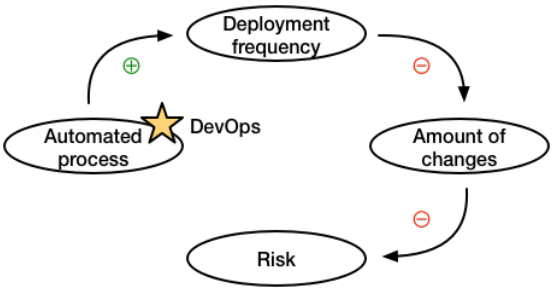

Fig. 1.2: DevOps practices help to break the vicious cycle through process automation

Investing in automation is not a new idea; many teams already write automated tests as part of their software development process. Practices such as test-driven development (TDD) [4] or continuous integration [13] — which will be discussed in more detail in chapter 6 — are common and widely accepted in the development community. This focus on test automation, along with the creation of multidisciplinary teams, helped to break the barrier between developers, testers and business analysts, creating a culture of collaboration between people with complimentary skills working as part of the same team.

Inspired by the success of Agile methods, a new movement emerged to take the same line of reasoning to the next level: the **DevOps** movement. Its goal is to create a culture of collaboration between development and operation teams that can increase the flow of completed work — higher frequency of deploys — while increasing the stability and reliability of the production environment.

Besides being a cultural change, the DevOps movement focuses a lot more on practical automation of the various activities necessary to tackle the last mile and deliver quality code to production, such as: code compilation, automated testing, packaging, creating environments for testing or production, infrastructure configuration, data migration, monitoring, log and metrics aggregation, auditing, security, performance, deployment, among others.

Companies that have implemented these DevOps practices successfully no longer see the IT department as a bottleneck but as an enabler to the busi-

ness. They can adapt to market changes quickly and perform several deploys per day safely. Some of them even make a new developer conduct a deploy to production on their first day of work!

This book will present, through actual examples, the main practices of DevOps and Continuous Delivery to allow you to replicate the same success in your company. The main objective of the book is to bring together the development and operations communities. Developers will learn about the concerns and practices involved in operating and maintaining stable systems in production, while system engineers and administrators will learn how to introduce changes in a safe and incremental way by leveraging automation.

1.3 About the book

The main objective of the book is to show how to apply DevOps and Continuous Delivery concepts and techniques in practice. For this reason, we had to choose which technologies and tools to use. Our preference was to use open source languages and tools, and to prioritize those that are used widely in industry.

You will need to use the chosen tools to follow the code examples. However, whenever a new tool is introduced, we will briefly discuss other alternatives, so that you can find the option that makes the most sense in your context.

You do not need any specific prior knowledge to follow the examples. If you have experience in the Java or Ruby ecosystem, that will be a bonus. Our production environment will run on UNIX (Linux, to be more specific), so a little experience using the command line can help but is not mandatory [12]. Either way, you will be able to run all the examples on your own machine, regardless if you are running Linux, Mac or Windows.

Target audience

This book is written for developers, system engineers, system administrators, architects, project managers and anyone with technical knowledge who has an interest in learning more about DevOps and Continuous Delivery practices.

Chapter structure

The book was written to be read from beginning to end sequentially. Depending on your experience with the topic of each chapter, you may prefer to skip a chapter or follow a different order.

Chapter 2 presents the sample application that will be used through the rest of the book and its technology stack. As the book's focus is not on the development of the application itself, we use a nontrivial application written in Java built on top of common libraries in the Java ecosystem. At the end of the chapter, the application will be running in production.

With the production environment running, in **chapter 3** we wear the operations team's hat and configure a monitoring server to detect failures and send notifications whenever a problem is encountered. At the end of the chapter, we will be notified that one of the servers crashed.

In **chapter 4** we will rebuild the problematic server, this time using automation and treating infrastructure as code. **Chapter 5** is a continuation of the subject, covering more advanced topics and refactoring the code to make it more modular, readable and extensible.

After wearing the operations team's hat, we will turn our attention to the software development side. **Chapter 6** discusses the Agile engineering practices that help writing quality code. You will learn about the various types of automated tests and launch a new server dedicated to perform continuous integration of our application code.

Chapter 7 introduces the concept of a deployment pipeline. We setup continuous integration for the infrastructure code, and implement an automated process to deploy the newest version of the application in production with the click of a button.

In **chapter 8**, we migrate the production environment to the cloud. Finally, we discuss more advanced topics, including resources for you to research and learn more about the topics that were left out of the scope of this book.

Code conventions

In the code examples, we will use ellipsis `...` to omit the unimportant parts. When making changes to already existing code, we will repeat the

lines around the area that needs to be changed to give more context about the change.

When the line is too long and does not fit on the page, we will use a backslash \ to indicate that the next line in the book is a continuation of the previous line. You can simply ignore the backslash and continue typing in the same line.

In the command line examples, in addition to the backslash, we use a greater than signal > in the next line to indicate that it is still part of the same command. This is to distinguish between the command that is being entered and the output produced when the command executes. When you run those commands, you can simply type it all in one line, ignoring the \ and the >.

More resources

We have created a discussion forum on Google Groups where you can send questions, suggestions, or feedback directly to the author. To subscribe, visit the URL:

https://groups.google.com/d/forum/devops-in-practice-book

All code examples from the book are also available on the author's GitHub projets, in these URLs:

https://github.com/dtsato/loja-virtual-devops

https://github.com/dtsato/loja-virtual-devops-puppet

Chapter 2

Everything starts in production

Contrary to what many software development processes suggest, the software life cycle should only begin when real users start using it. The last mile problem presented in chapter 1 should really be the first mile, because no software delivers value before going into production. Therefore, the objective of this chapter is to launch a complete production environment, starting from scratch, and install a relatively complex Java application that will be used as a starting point for the DevOps concepts and practices presented in the rest of the book.

By the end of this chapter, we will have a complete e-commerce application running – backed by a database – which will allow users to register and make purchases, in addition to providing administrative tools to manage the product catalog, promotions and the static content available at the online store.

What is the point of adding new features, improving performance, fixing

bugs, creating beautiful screens for the website if it is not in production? Let's start by tackling the most difficult part, the last mile and put the software into production.

2.1 Our example application: an online store

As the book's focus is not on the development process itself, but rather on the DevOps practices that help to build, deploy and operate an application in production, we will use a sample application based on an open source project. The online store is a web application written in Java, using the *Broadleaf Commerce* platform (http://www.broadleafcommerce.org/) . The code used in this chapter and throughout the book was written based on the demo site created by Broadleaf and can be accessed at the following GitHub repository:

https://github.com/dtsato/loja-virtual-devops/

Broadleaf Commerce is a flexible platform that provides configuration and cusomtization points to extend its functionality. However, it already implements many of the standard features of an online shopping website such as Amazon, including:

- Product catalog browsing and search;
- Product pages with name, description, price, photos, and related products;
- Shopping cart;
- Customizable checkout process including promotions, payment and shipping information;
- User registration;
- Order history;
- Administration tools for managing: the product catalog, promotions, prices, shipping rates and pages with customized content.

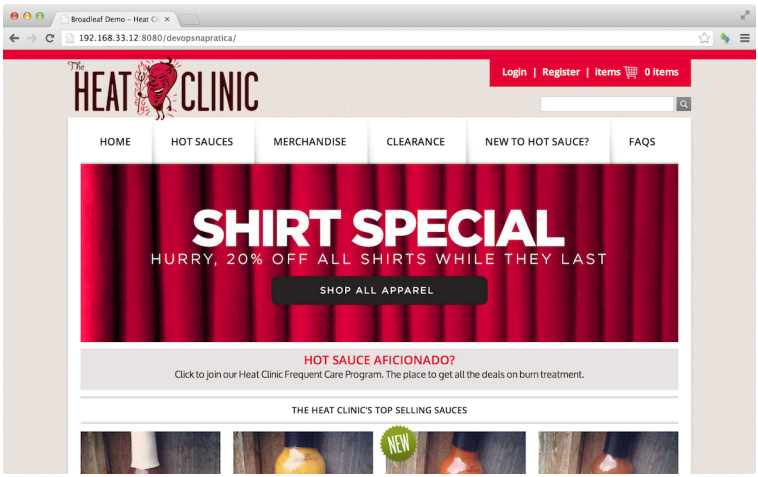

Fig. 2.1: A preview of the online store's homepage

Moreover, the *Broadleaf Commerce* platform is built using several well established frameworks in the Java community, making it an interesting example from a DevOps perspective, because it is a good representation of the complexity involved in the build and deploy process of a Java application in the real world. The implementation details are beyond the scope of this book, but it is important to have an overview of the technologies and frameworks used in this application:

- **Java**: The application is written in Java (http://java.oracle.com/) , compatible with Java SE 6 and above.
- **Spring**: Spring (http://www.springframework.org/) is a popular framework for Java enterprise applications that offers several components such as: dependency injection, transaction management, security, an MVC framework, among others.
- **JPA e Hibernate**: JPA is the Java Persistence API and Hibernate (http://www.hibernate.org/) is the most popular JPA implementation in Java community, allowing developers to perform object-relational mapping (ORM) between Java objects and database tables.

- **Google Web** Toolkit: GWT (http://developers.google.com/web-toolkit/) is a framework written by Google to facilitate the creation of rich interfaces that run in the browser. It allows the developer to write Java code that is then compiled to Javascript. The online store uses GWT to implement the administration UI.

- **Apache Solr**: Solr (http://lucene.apache.org/solr/) is a search server that allows indexing of the product catalog and the online store and offers a powerful and flexible API to query full text searches throughout the catalog.

- **Tomcat**: Tomcat (http://tomcat.apache.org/) is a server that implements the web components of Jave EE – Java Servlet and Java Server Pages (JSP). Although *Broadleaf Commerce* runs on alternative application servers – such as Jetty, GlassFish or JBoss – we will use Tomcat because it is a common choice for many enterprises running Java web applications.

- **MySQL** : MySQL (http://www.mysql.com/) is a relational database server. JPA and Hibernate allow the application to run on several other database servers – such as Oracle, PostgreSQL or SQL Server – but we will also use MySQL because it is open source and is also a popular choice.

This is not a small application. Better yet, it uses libraries that are often found in the majority of real world Java applications. As we have mentioned, we will use it as our example, but you can also follow the process with your own application, or even choose another software, regardless of language, in order to learn the DevOps techniques that will be presented and discussed in the book.

Our next objective is to make the online store live. Before making the first deploy we must have a production environment ready with servers where the code can run. The production environment in our example will initially be composed of two servers, as shown in figure 2.2.

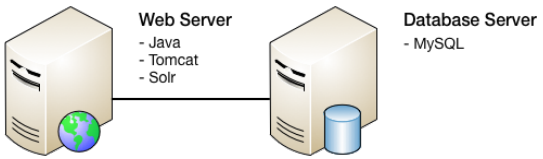

Fig. 2.2: Production environment for the online store

Users will access the online store through a web server, which will run an instance of Tomcat. The web server runs all libraries and Java frameworks used by the online store, including an embedded Solr instance. Finally, the web application will use MySQL, running on a separate database server. This is a two-tier architecture commonly used by several real world applications. In chapter 8 we will discuss in more detail the factors that influence the choice of your application's physical architecture, but for now, let's follow a common pattern to simplify the example and make the store live as soon as possible.

2.2 Installing the production environment

Buying servers and hardware for the online store would cost too much money, so we will initially use virtual machines (VM). This allows us to run the entire production environment on our machine. There are several tools available for running virtual machines – such as VMware or Parallels – however some are paid and most use a GUI for configuration, which would turn this chapter into a collection screenshots that are hard to follow.

To solve these problems, we will use two tools that simplify running and configuring virtual machines: **Vagrant** (http://www.vagrantup.com) and **VirtualBox** (http://www.virtualbox.org) . VirtualBox is Oracle's VM hypervisor that lets you configure and run virtual machines on all major platforms: Windows, Linux, Mac OS X and Solaris. To further simplify our task, we will also use Vagrant, that provides a Ruby DSL to define, manage and configure virtual environments. Vagrant also has a simple command line interface for interacting with these virtual environments.

If you already have Vagrant and VirtualBox installed and running, you can skip to subsection *"Declaring and booting the servers"*. Otherwise, the

installation process for these tools is simple.

Installing VirtualBox

To install VirtualBox, visit the downloads page: http://www.virtualbox.org/wiki/Downloads and select the latest version. At the time of writing, the newest version is VirtualBox 4.3.8 and the examples will use it throughout this book. Select the installation package according to your platform: on Windows, the installer is an executable `.exe` file; On Mac OS X, the installer is a `.dmg` package; on Linux, VirtualBox offers both `.deb` or `.rpm` packages depending on your distribution.

Once you have downloaded the package, install it: on Windows and Mac OS X, just double-click on the installation file (`.exe` on Windows and `.dmg` on Mac OS X) and follow the installer's instructions. On Linux, if you have chosen the `.deb` package, install it by running the `dpkg -i {file.deb}` command, replacing `{file.deb}` with the name of the file you have downloaded, e.g. `virtualbox-4.3_4.3.8-92456~Ubuntu~raring_i386.deb`.
If you have chosen the `.rpm` package, install it by executing the `rpm -i {file.rpm}` command, replacing `{file.rpm}` with the name of the file you have downloaded, e.g. `VirtualBox-4.3-4.3.8_92456_el6-1.i686.rpm`.

Note: On Linux, if your user is not `root`, you will need to execute the previous commands with the `sudo` command in front, for example: `sudo dpkg -i {file.deb}` or `sudo rpm -i {file.rpm}`.

To test that VirtualBox is installed properly, go to the console and run the `VBoxManage -v` command. If you are running on Windows, to open the console you can use `Win+R` and type `cmd`. If everything is correct, the command will run and print an output like "4.3.8r92456", depending on the installed version.

Installing Vagrant

Once VirtualBox is installed, we can continue with Vagrant's installation process, which is very similar. Visit Vagrant's downloads page http://www.vagrantup.com/downloads.html and select the latest version. At the

time of writing, the newest version is Vagrant 1.5.1 and the examples will use it throughout the book. Select the installation package according to your platform. On Windows, the installer is an `.msi` file; on Mac OS X, the installer is a `.dmg` package; on Linux, Vagrant offers both `.deb` or `.rpm` packages depending on your distribution.

Once you have downloaded the package, install it. On Windows and on Mac OS X, just do a double-click the installation file (`.msi` on Windows and `.dmg` on Mac OS X) and follow the installer's instructions. On Linux, just follow the same steps of the Linux VirtualBox installation using the chosen package (`vagrant_1.5.1_i686.deb` or `vagrant_1.5.1_i686.rpm`). On Mac OS X and Windows, the command `vagrant` is already added to the `PATH` after installation. On Linux, you will need to add `/opt/vagrant/bin` to your `PATH` manually.

To test that Vagrant is installed properly, go to the console and run the `vagrant -v` command. If everything is correct, the command will run and print an output like "Vagrant 1.5.1", depending on the installed version.

The last thing you need to do is to set up an initial image that serves as a template for initializing new VMs. These images, also known as *box*, serve as a starting point, containing the base operating system. In our case we will use a *box* offered by Vagrant, which contains the image of a 32-bit Linux Ubuntu 12.04 LTS. To download and configure this *box*, you must run the command:

```
$ vagrant box add hashicorp/precise32
==> box: Loading metadata for box 'hashicorp/precise32'
    box: URL: https://vagrantcloud.com/hashicorp/precise32
...
```

This command will download a large VM image file (299MB), so it will require a good internet connection and will take some time to finish. There are many other images made available by the Vagrant community that can be found at http://www.vagrantbox.es/ and the setup process is similar to the previous command. The company behind Vagrant, HashiCorp, recently introduced the **Vagrant Cloud** (https://vagrantcloud.com/) , as a way to simplify the process of finding and sharing *"boxes "* with the community.

Declaring and booting the servers

Once Vagrant and VirtualBox are installed and working, it is simple to declare and manage a virtual environment. Our VM configuration is declared in a file called `Vagrantfile`. The `vagrant init` command creates an initial `Vagrantfile`, with comments explaining all the available configuration options.

In our case, we will start simple and only set up what is required for our two production servers. The content we need in our `Vagrantfile` is:

```
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  config.vm.box = "hashicorp/precise32"

  config.vm.define :db do |db_config|
    db_config.vm.hostname = "db"
    db_config.vm.network :private_network,
                         :ip => "192.168.33.10"
  end

  config.vm.define :web do |web_config|
    web_config.vm.hostname = "web"
    web_config.vm.network :private_network,
                          :ip => "192.168.33.12"
  end
end
```

This will configure two VMs, named `db` and `web`. Both use the `hashicorp/precise32` *box* that we have installed earlier. Each has a configuration block defining their hostname and an IP address for network connectivity. The IP addresses `192.168.33.10` and `192.168.33.12` were chosen arbitrarily to avoid conflicts with the network settings of your machine. To launch the servers, just execute the `vagrant up` command in the same directory where the `Vagrantfile` file was created, and Vagrant will take care of booting and configuring the virtual machines for you. If all goes well, the command output will look like:

```
$ vagrant up
Bringing machine 'db' up with 'virtualbox' provider...
Bringing machine 'web' up with 'virtualbox' provider...
==> db: Importing base box 'hashicorp/precise32'...
==> db: Matching MAC address for NAT networking...
==> db: Checking if box 'hashicorp/precise32' is up to date...
==> db: Setting the name of the VM: blank_db_1394680068450_7763
==> db: Clearing any previously set network interfaces...
==> db: Preparing network interfaces based on configuration...
    db: Adapter 1: nat
    db: Adapter 2: hostonly
==> db: Forwarding ports...
    db: 22 => 2222 (adapter 1)
==> db: Booting VM...
==> db: Waiting for machine to boot. This may take a few
                                                        minutes...
    db: SSH address: 127.0.0.1:2222
    db: SSH username: vagrant
    db: SSH auth method: private key
==> db: Machine booted and ready!
==> db: Configuring and enabling network interfaces...
==> db: Mounting shared folders...
    db: /vagrant => /private/tmp/blank
==> web: Importing base box 'hashicorp/precise32'...
==> web: Matching MAC address for NAT networking...
==> web: Checking if box 'hashicorp/precise32' is up to date...
==> web: Setting the name of the VM:
                                    blank_web_1394680087001_3842
==> web: Fixed port collision for 22 => 2222. Now on port 2200.
==> web: Clearing any previously set network interfaces...
==> web: Preparing network interfaces based on configuration...
    web: Adapter 1: nat
    web: Adapter 2: hostonly
==> web: Forwarding ports...
    web: 22 => 2200 (adapter 1)
==> web: Booting VM...
==> web: Waiting for machine to boot. This may take a few
                                                         minutes...
    web: SSH address: 127.0.0.1:2200
```

```
    web: SSH username: vagrant
    web: SSH auth method: private key
==> web: Machine booted and ready!
==> web: Configuring and enabling network interfaces...
==> web: Mounting shared folders...
    web: /vagrant => /private/tmp/blank
```

This means the virtual machines are running and ready to be used. If you read the output carefully, you will notice that Vagrant executed similar steps for both `db` and `web` machines. Vagrant has also mapped SSH ports and setup a partition to allow the virtual machine to access files from the working directory on the host machine. This may sound a bit confusing, but it demonstrates some of the powerful features offered by Vagrant.

2.3 Configuring the production servers

Now that the servers are running, you need to install the packages and dependencies so that the online store can function properly. System administrators are probably already familiar with working on several machines at the same time, but developers often work on a single machine.

In the next sections, we will begin to interact with more than one machine simultaneously, and the commands will need to be executed on the correct server. We will use a special notation for the instructions in the rest of the book, representing the user and the server in the command line prompt to serve as a reminder to check if you are running the command in the right server. The notation will be `{user}@{server}$`, in which the user will usually be `vagrant` and the server may vary between `db` or `web`. Thus, commands preceded by `vagrant@db$` should be run on the `db` server and those preceded by `vagrant@web$` should be run on the `web` server.

Instead of jumping from one server to another, we will initially focus on one server at a time, starting with the database server.

Database server

First of all, we need to login to the database server to install MySQL. This is done with the `vagrant ssh db` command, which opens an SSH session

with the `db` server. Once logged in, we can install MySQL server using the `mysql-server` package:

```
vagrant@db$ sudo apt-get update
Ign http://us.archive.ubuntu.com precise InRelease
Ign http://us.archive.ubuntu.com precise-updates InRelease
...
Reading package lists... Done
vagrant@db$ sudo apt-get install mysql-server
Reading package lists... Done
Building dependency tree
...
ldconfig deferred processing now taking place
```

To understand what is happening, we will analyze each part of this command. The `apt-get` command is used to manage which packages are installed in the system. The `update` command tells APT to update its package index to the latest version. The `install` command is used to install a new package on the system. In this case, the package we want to install is `mysql-server`. The entire command is preceded by the `sudo` command. On UNIX systems, there is a special user with "super powers" known as `root`. The `sudo` command allows you to run something as if the current user was `root`.

The MySQL server installation process requires you to choose a password for the `root` user. You will need to provide this password twice during the installation process. The choice of a strong password is important to improve the system security, however, it is also a separate topic of discussion, which we will cover in chapter 8. A malicious user that discovers the master password can run any command on the database, including deleting all the tables!

In real life you do not want this to happen. However, since this is just an example for the book, we will use a simple password to make it easy to remember: "secret." If you prefer, you may choose another password, but from now on be careful to replace any mention of the password "secret" with your chosen password.

When you finish installing the package, MySQL server will be running, but only for local access. Since we need to access the database

from the web server, you must configure MySQL to allow external connections. To do this, we will create a configuration file called `/etc/mysql/conf.d/allow_external.cnf`. You may use the text editor of your choice – `emacs` and `vim` are two common options – however we will use the `nano` editor because it is already installed and it is simpler to learn for those who are not familiar with editing files on the command line. The following command creates the configuration file as `root`:

```
vagrant@db$ sudo nano /etc/mysql/conf.d/allow_external.cnf
```

Then type the content of the file below and use `Ctrl+O` to save and `Ctrl+X` to save and exit the editor:

```
[mysqld]
   bind-address = 0.0.0.0
```

Once the file has been created, you must restart MySQL server for it to use the new configuration. This is done with the command:

```
vagrant@db$ sudo service mysql restart
mysql stop/waiting
mysql start/running, process 6460
```

At this moment, MySQL server is already up and running, however we have not created any databases yet. It is common for each application to create its own database or schema, to persist its tables, indexes and data. In order to create a database for the online store, which we will call the `store_schema`, we will execute the command:

```
vagrant@db$ mysqladmin -u root -p create store_schema
Enter password:
```

When executing this command, the `-u root` option instructs that the connection with MySQL should be made as the `root` user and the `-p` option will ask you for the user's password which, in this case, will be "secret." The rest of the command is self-explanatory. To verify that the database was successfully created, you may run the following command that lists all existing databases:

```
vagrant@db$ mysql -u root -p -e "SHOW DATABASES"
Enter password:
+--------------------+
| Database           |
+--------------------+
| information_schema |
| store_schema       |
| mysql              |
| performance_schema |
| test               |
+--------------------+
```

In the same way it is not recommended to run commands on the operating system as `root`, it is also not recommended to run queries in MySQL as `root`. A good practice is to create specific users for each application, granting it only the necessary access to run the application. Besides that, the standard MySQL server installation includes an anonymous account for database access. Before creating the online store account, we need to remove the anonymous account:

```
vagrant@db$ mysql -uroot -p -e "DELETE FROM mysql.user WHERE \
> user=''; FLUSH PRIVILEGES"
Enter password:
```

Once the anonymous account is removed, we will create a user called `store` with a password "storesecret", with exclusive access to the `store_schema` schema:

```
vagrant@db$ mysql -uroot -p -e "GRANT ALL PRIVILEGES ON \
> store_schema.* TO 'store'@'%' IDENTIFIED BY 'storesecret';"
Enter password:
```

In order to make sure that the new user was properly created, we can execute a simple query to verify that everything is OK (the password should be "storesecret" because we are running the command as the `store` user instead of `root`):

```
vagrant@db$ mysql -u store -p store_schema -e \
> "select database(), user()"
```

```
Enter password:
+---------------+-----------------+
| database()    | user()          |
+---------------+-----------------+
| store_schema  | store@localhost |
+---------------+-----------------+
```

At this point, the database server is configured and running correctly. To logout from the `db` server, you can type the `logout%` command or press `Ctrl+D`.

The database is still empty – no tables or data were created – but we will take care of that in the 2.4 section. Before we can do that, however, we must finish setting up the rest of the infrastructure: the web server.

Web server

We are now going to install Tomcat and configure a data source to use the database we have just created. For this, we need to login by running the `vagrant ssh web` command, which opens an SSH session with the `web` server.

Once logged in, we can install Tomcat using the `tomcat7` package. We also need to install a client to connect to the MySQL database, available in the `mysql-client` package:

```
vagrant@web$ sudo apt-get update
Ign http://us.archive.ubuntu.com precise InRelease
Ign http://us.archive.ubuntu.com precise-updates InRelease
...
Reading package lists... Done
vagrant@web$ sudo apt-get install tomcat7 mysql-client
Reading package lists... Done
Building dependency tree
...
ldconfig deferred processing now taking place
```

This will install Tomcat 7, MySQL client and all its dependencies, including Java 6. In order to verify that Tomcat is running properly, you can open your browser and visit the URL http://192.168.33.12:8080/ – `192.168.33.12`

is the IP address configured in Vagrant for the `web` server and `8080` is the default Tomcat port. If everything is ok, you will see a page with the message *"It works!"*.

The online store requires a secure connection with the server to transfer personal data – such as passwords or credit card numbers – in an encrypted form. For that, we need to configure an SSL connector so that Tomcat may run on both HTTP and HTTPS. SSL connections use a certificate signed by a trusted authority to identify the server and to encrypt data transferred between your browser and the server. We will use the `keytool` command, which is part of the standard Java distribution, to generate a certificate for our web server. During this process, you will be prompted to provide information about the server along with a password to protect the `keystore`. In our case, we will use "secret" again for simplicity. The full command, along with an example of the information you must supply, is:

```
vagrant@web$ cd /var/lib/tomcat7/conf
vagrant@web$ sudo keytool -genkey -alias tomcat -keyalg RSA \
> -keystore .keystore
Enter keystore password:
Re-enter new password:
What is your first and last name?
  [Unknown]:  Online Store
What is the name of your organizational unit?
  [Unknown]:  Devops in Practice
What is the name of your organization?
  [Unknown]:  Code Crushing
What is the name of your City or Locality?
  [Unknown]:  Chicago
What is the name of your State or Province?
  [Unknown]:  IL
What is the two-letter country code for this unit?
  [Unknown]:  US
Is CN=Online Store, OU=Devops in Practice, ..., C=US correct?
  [no]:  yes

Enter key password for <tomcat>
  (RETURN if same as keystore password):
```

Explaining the command step by step: the `-genkey` option generates a new key; the `-alias tomcat` option defines an alias for the certificate; the `-keyalg RSA` option specifies the algorithm used to generate the key, in this case RSA; and finally, the `-keystore .keystore` option determines the file where the `keystore` will be saved. This file format allows multiple keys/certificates to be stored in the same `keystore`, so it is good to choose an alias for the certificate generated for Tomcat. Besides, both the `keystore` and Tomcat's certificate are password protected. In this case, we will again use "secret" for both.

Once the certificate has been created and stored, we need to configure the Tomcat server to enable the SSL connector. This is done by editing the file `/var/lib/tomcat7/conf/server.xml`:

```
vagrant@web$ sudo nano /var/lib/tomcat7/conf/server.xml
```

In this file, the definition of the SSL connector for port 8443 is already predefined, however it is commented out. You need to uncomment the definition for this connector and add two more attributes that represent the `keystore` file and the password protection. The relevant part of the XML file that you need to change is:

```
<?xml version='1.0' encoding='utf-8'?>
<Server port="8005" shutdown="SHUTDOWN">
  ...
  <Service name="Catalina">
    ...
    <!-- Define a SSL HTTP/1.1 Connector on port 8443
     This connector uses the JSSE configuration, when using
     APR, the connector should be using the OpenSSL style
     configuration described in the APR documentation -->

    <Connector port="8443" protocol="HTTP/1.1"
               SSLEnabled="true"
               maxThreads="150" scheme="https" secure="true"
               keystoreFile="conf/.keystore"
               keystorePass="secret"
               clientAuth="false" sslProtocol="SSLv3" />
    ...
```

```
  </Service>
</Server>
```

The web server configuration is almost finished. The last change we need to do is to increase the amount of memory that Tomcat may use. The online store requires more memory than the default settings for Tomcat, so we need to change a line in the `/etc/default/tomcat7` file:

```
vagrant@web$ sudo nano /etc/default/tomcat7
```

The line that defines the `JAVA_OPTS` variable needs to be changed to allow the Java virtual machine to use up to 512Mb of memory, instead of the default 128Mb. After this change, the `JAVA_OPTS` variable should look like:

```
JAVA_OPTS="-Djava.awt.headless=true -Xmx512M
                                    -XX:+UseConcMarkSweepGC"
```

Finally, we need to restart Tomcat so that all settings are applied:

```
vagrant@web$ sudo service tomcat7 restart
```

To test that everything is working, you can return to your browser and try to access the home page through the SSL connector by visiting the URL https://192.168.33.12:8443/. Depending on which browser you are using, a warning will pop up saying that the server certificate was self-signed. In a real life scenario, you would usually buy an SSL certificate signed by a trusted authority, but in this case we will continue and accept the warning because we know that the certificate was intentionally self-signed by us.

If all goes well, you will again see the *"It works!"* message, but this time using a secure HTTPS connection.

At this point, both servers are configured and ready for the final step: To build and deploy the application, and make the online store live!

2.4 Application build and deploy

Up until now, the database and web server configuration was fairly generic. We installed the basic software packages to run MySQL and Tomcat, we configured HTTP/HTTPS connectors, we created a user and an empty database

which are still not directly associated with the online store except for the *"schema"* and the user name we chose.

The time to download and compile the code, run the tests, package the application and put the online store live has arrived! This process of compiling, testing and packaging is known as the *"build "* process. The build process may also include steps for managing dependencies, running static code analysis tools, calculating code coverage, generating documentation, among others. The main goal of the build process is to create one or more artifacts, with specific versions, that can become *"release candidates"* to be deployed in production.

It is generally recommended to run the building process in a production-like environment in order to avoid incompatibilities between operating systems or library versions. We could even set up Vagrant to create a new VM to be used as a dedicated build server. However, to simplify the process in this chapter, we will run the building process in our web server, which is where the deploy will happen. This will save us many steps now, but we will revisit this process in chapter 6.

To run the build, you need to install a few more tools: **Git** (http://git-scm.com/) for version control, **Maven** (http://maven.apache.org/) to run the build itself and the **JDK** (Java Development Kit) to compile our Java code:

```
vagrant@web$ sudo apt-get install git maven2 openjdk-6-jdk
Reading package lists... Done
Building dependency tree
...
ldconfig deferred processing now taking place
```

Once these tools are installed, we need to download the code hosted on Github using the `git clone` command and run the build with the `mvn install` command:

```
vagrant@web$ cd
vagrant@web$ git clone \
> https://github.com/dtsato/loja-virtual-devops.git
Cloning into 'loja-virtual-devops'...
remote: Counting objects: 11358, done.
remote: Compressing objects: 100% (3543/3543), done.
```

```
remote: Total 11358 (delta 6053), reused 11358 (delta 6053)
Receiving objects: 100% (11358/11358), 55.47 MiB | 1.14 MiB/s,
                                                          done.
Resolving deltas: 100% (6053/6053), done.
vagrant@web$ cd loja-virtual-devops
vagrant@web$ export MAVEN_OPTS=-Xmx256m
vagrant@web$ mvn install
[INFO] Scanning for projects...
[INFO] Reactor build order:
[INFO]   loja-virtual-devops
[INFO]   core
[INFO]   admin
[INFO]   site
[INFO]   combined
[INFO] ------------------------------------------------------
[INFO] Building loja-virtual-devops
[INFO]    task-segment: [install]
[INFO] ------------------------------------------------------
...
```

The first Maven run will take about 20 to 25 minutes, during which time several things will happen:

- **Sub-projects**: the project is organized into 4 modules: `core` holds the main settings, extensions and customizations of the online store; `site` is the online store web application; `admin` is a restricted web app for administrating and managing the online store; finally, `combined` is a module that combines the web application and the `core` into a single artifact. Each module acts as a sub-project in which Maven needs to perform a separate build.

- **Dependency resolution**: Maven will resolve all project dependencies and download the frameworks and libraries required to compile and run each module of the online store.

- **Compilation**: Java and GWT code needs to be compiled for execution. It takes a long time when it is first compiled. The Java compiler is smart enough to recompile only what is necessary in subsequent builds.

- **Automated testing**: In modules that include automated tests, Maven will execute them and generate reports about which tests have passed and which ones have failed.

- **Packaging artifacts**: finally, the `core` module will be packaged as a `jar`% archive while the web modules (`site`, `admin` and `combined`) will be packaged as a `.war` file. Both `jar` and `war` files are equivalent to the `zip` files you regularly download from the internet, but the internal structure of the packages is well defined and standardized by Java: the `jar` contains compiled classes and serves as a library while the `war` contains classes, libraries, static assets and configuration files needed to run a web application in a Java container such as Tomcat.

When the build is finished, if all goes well, you should see a Maven output that looks like:

```
...
[INFO] ------------------------------------------------------
[INFO] Reactor Summary:
[INFO] ------------------------------------------------------
[INFO] loja-virtual-devops ............... SUCCESS [1:35.191s]
[INFO] core .............................. SUCCESS [6:51.591s]
[INFO] admin ............................. SUCCESS [6:54.377s]
[INFO] site .............................. SUCCESS [4:44.313s]
[INFO] combined .......................... SUCCESS [25.425s]
[INFO] ------------------------------------------------------
[INFO] ------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------
[INFO] Total time: 20 minutes 31 seconds
[INFO] Finished at: Sat Mar 15 02:34:24 UTC 2014
[INFO] Final Memory: 179M/452M
[INFO] ------------------------------------------------------
```

This means that the build ran successfully and the artifact that we are interested in deploying is the `devopsnapratica.war` file of the `combined` module, which can be found in the `combined/target` directory.

Before you deploy the `war` file, the last setting required by Tomcat is to define our data sources, or `DataSource`, that the application will use to connect to the database. If such configuration was built within the artifact itself, we would need to execute a new build every time the connection parameters to the database changed or if we wanted to run the same application in different environments. Instead, the application depends on abstract JNDI resources (Java Naming and Directory Interface). These resources need to be configured and exposed by the container, allowing them to change without requiring a new build of the application.

The online store needs three data sources in JNDI called `jdbc/web`, `jdbc/secure` and `jdbc/storage`. In our case, we will use the same database for all three resources, defined in the `context.xml` file:

```
vagrant@web$ sudo nano /var/lib/tomcat7/conf/context.xml
```

The relevant settings are the three `<Resource>` elements and the difference between them is only the `name` attribute:

```
<?xml version='1.0' encoding='utf-8'?>
<Context>
  <WatchedResource>WEB-INF/web.xml</WatchedResource>

  <Resource name="jdbc/web" auth="Container"
      type="javax.sql.DataSource" maxActive="100" maxIdle="30"
      maxWait="10000" username="store" password="storesecret"
      driverClassName="com.mysql.jdbc.Driver"
      url="jdbc:mysql://192.168.33.10:3306/store_schema"/>

  <Resource name="jdbc/secure" auth="Container"
      type="javax.sql.DataSource" maxActive="100" maxIdle="30"
      maxWait="10000" username="store" password="storesecret"
      driverClassName="com.mysql.jdbc.Driver"
      url="jdbc:mysql://192.168.33.10:3306/store_schema"/>

  <Resource name="jdbc/storage" auth="Container"
      type="javax.sql.DataSource" maxActive="100" maxIdle="30"
      maxWait="10000" username="store" password="storesecret"
      driverClassName="com.mysql.jdbc.Driver"
```

```
      url="jdbc:mysql://192.168.33.10:3306/store_schema"/>
</Context>
```

To deploy the online store, we simply need to copy the `war` artifact to the right place and Tomcat will detect it:

```
vagrant@web$ cd ~/loja-virtual-devops
vagrant@web$ sudo cp combined/target/devopsnapratica.war \
> /var/lib/tomcat7/webapps
```

In the same way that the build process took a while, the deploy process will also take a few minutes for the first time, since the application will create the tables and populate the database using Hibernate. You can monitor the deploy process looking at Tomcat logs. The `tail -f` command is useful in this case because it shows in real time everything that is being appended to the log:

```
vagrant@web$ tail -f /var/lib/tomcat7/logs/catalina.out
Mar 15, 2014 2:08:51 AM org.apache.coyote.AbstractProtocol start
INFO: Starting ProtocolHandler ["http-bio-8080"]
Mar 15, 2014 2:08:51 AM org.apache.coyote.AbstractProtocol start
INFO: Starting ProtocolHandler ["http-bio-8443"]
Mar 15, 2014 2:08:51 AM org.apache.catalina.startup.Catalina
                                                          start
INFO: Server startup in 408 ms
...
```

The deploy will be finished as soon as you realize that the activities in the log have stopped. To exit the `tail` command you can use `Ctrl+C`. Now just test that everything is working. Open a new browser window and visit the URL http://192.168.33.12:8080/devopsnapratica/.

Congratulations— the online store is in production!

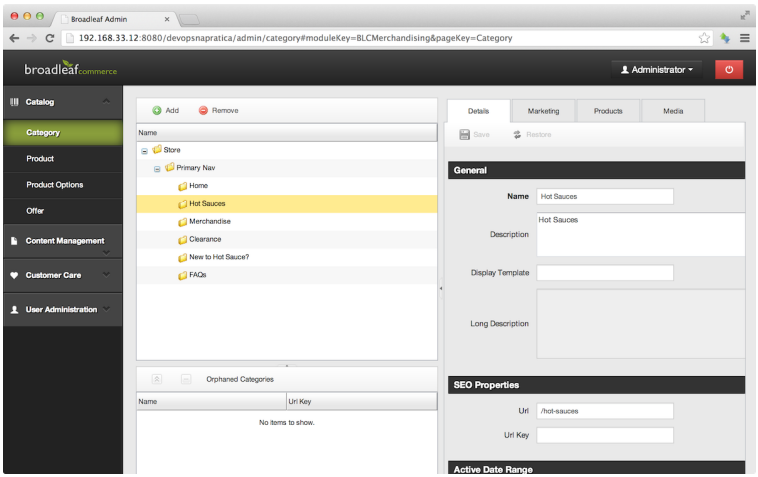

Fig. 2.3: Administration page of the online store

You can also access the administration page visiting the URL http://192.168.33.12:8080/devopsnapratica/admin/ (the slash at the end of the URL is important). Using the "admin" user and "admin" password credentials, you can browse the administration pages and manage the product catalog, its categories, the content of the online store pages, orders, as well as users and permissions.

Now that we are in production and we conquered the last mile, we can focus on the DevOps principles and practices required to operate our application in production, while enabling further changes to be made reliably.

Chapter 3

Monitoring

Now that the online store is in production, we conquered the most important part of any software life cycle. However, this was only the first step! From now on, we must wear the operations team's hat and be responsible for the stability of the production system: we need to ensure that the online store will continue running smoothly.

How do you know if the system is functioning normally or if it is offline? You may access the shop manually and check if everything is ok, but it takes time and relies on you remembering to do it. You may also wait for user complaints, but this is not a good option either. Monitoring systems exist to solve this problem. You define what are the necessary checks to know if the system is working and the monitoring system will verify them and send you a notification as soon as any check fails.

A good monitoring system will alert you when something is wrong, allowing you to investigate and quickly fix what is causing the problem. This

way, you do not have to rely on your users to warn you when something is offline, enabling a much more proactive approach to resolve production issues.

In this chapter we will install a monitoring system and configure it with various checks to assure that the online store is running in production. We will use one of the most popular monitoring tools used by operations teams: **Nagios** (http://www.nagios.org/) . Although it is an old tool and its graphical interface is not very user friendly, it is still one of the most popular options because it has a robust implementation and a huge plugin ecosystem maintained by the community. By the end of the chapter, our production environment will include a new server, as shown in figure 3.1.

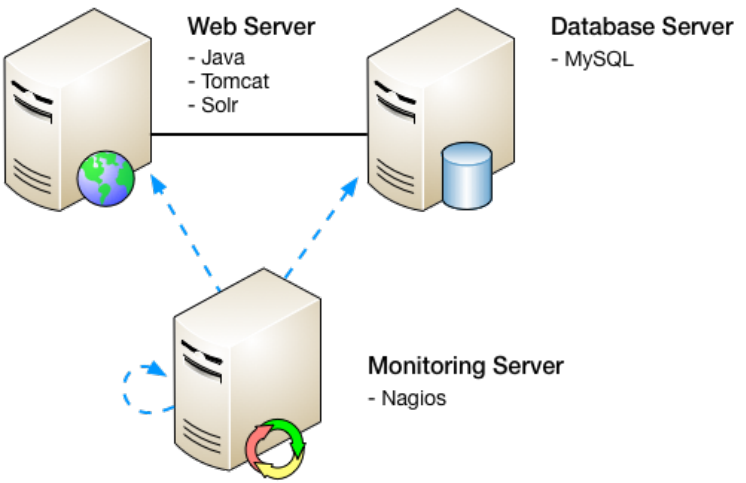

Fig. 3.1: Online store production environment

3.1 Installing the monitoring server

In a monitoring system, hosts usually operate in two main roles: *supervisor* or *supervised*. The **supervisor** is responsible for collecting information about the rest of the infrastructure, evaluating whether it is healthy and sending alerts when problems are found. The **supervised** are those hosts doing the actual work and that you are interested in monitoring. In our online store, the web server and database server will be supervised by a new server that we will

install and configure to act as the supervisor.

Running the supervisor externally to the supervised hosts is a good practice. If the supervisor is running on the same machine and a problem occurs, both will be unavailable and you may not be alerted about the problem. For this reason, we will declare a new virtual machine in our `Vagrantfile` called `monitor` and with an IP address `192.168.33.14`:

```
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  config.vm.box = "hashicorp/precise32"

  config.vm.define :db do |db_config|
    db_config.vm.hostname = "db"
    db_config.vm.network :private_network,
                         :ip => "192.168.33.10"
  end

  config.vm.define :web do |web_config|
    web_config.vm.hostname = "web"
    web_config.vm.network :private_network,
                          :ip => "192.168.33.12"
  end

  config.vm.define :monitor do |monitor_config|
    monitor_config.vm.hostname = "monitor"
    monitor_config.vm.network :private_network,
                              :ip => "192.168.33.14"
  end
end
```

After saving the file, just run `vagrant up` in the same directory to boot the new VM. If all goes well, the output of the command will look like:

```
$ vagrant up
Bringing machine 'db' up with 'virtualbox' provider...
Bringing machine 'web' up with 'virtualbox' provider...
Bringing machine 'monitor' up with 'virtualbox' provider...
==> db: Checking if box 'hashicorp/precise32' is up to date...
```

```
==> db: VirtualBox VM is already running.
==> web: Checking if box 'hashicorp/precise32' is up to date...
==> web: VirtualBox VM is already running.
==> monitor: Importing base box 'hashicorp/precise32'...
==> monitor: Matching MAC address for NAT networking...
==> monitor: Checking if box 'hashicorp/precise32' is
                                                    up to date...
==> monitor: Setting the name of the VM:
              blank_monitor_1394851889588_2267
==> monitor: Fixed port collision for 22 => 2222.
                                                    Now on port 2201.
==> monitor: Clearing any previously set network interfaces...
==> monitor: Preparing network interfaces based on
                                                    configuration...
    monitor: Adapter 1: nat
    monitor: Adapter 2: hostonly
==> monitor: Forwarding ports...
    monitor: 22 => 2201 (adapter 1)
==> monitor: Running 'pre-boot' VM customizations...
==> monitor: Booting VM...
==> monitor: Waiting for machine to boot. This may take a few
                                                          minutes...
    monitor: SSH address: 127.0.0.1:2201
    monitor: SSH username: vagrant
    monitor: SSH auth method: private key
    monitor: Error: Connection timout. Retrying...
==> monitor: Machine booted and ready!
==> monitor: Configuring and enabling network interfaces...
==> monitor: Mounting shared folders...
    monitor: /vagrant => /private/tmp/blank
```

To complete the basic installation, we need to login to the server by running the `vagrant ssh monitor` command. Once logged in, we can install Nagios using the `nagios3` package. However, before installing Nagios, we need to configure the system to use the latest Nagios version. For this, we have to run the following commands:

```
vagrant@monitor$ echo "Package: nagios*
> Pin: release n=raring
```

```
> Pin-Priority: 990" | sudo tee /etc/apt/preferences.d/nagios
Package: nagios*
Pin: release n=raring
Pin-Priority: 990
vagrant@monitor$ echo "deb \
> http://old-releases.ubuntu.com/ubuntu raring main" | \
> sudo tee /etc/apt/sources.list.d/raring.list
deb http://old-releases.ubuntu.com/ubuntu raring main
vagrant@monitor$ sudo apt-get update
Ign http://old-releases.ubuntu.com raring InRelease
Ign http://security.ubuntu.com precise-security InRelease
...
```

This adds a new entry to the package sources list with the newest Ubuntu release – called "raring" – and sets the preferences in the package installer to give priority to the latest release on Nagios packages. The `apt-get update` command tells the package manager to update its index so the new versions become available. Once this is done, we may install nagios with the command:

```
vagrant@monitor$ sudo apt-get install nagios3
Reading package lists... Done
Building dependency tree
...
```

Nagios depends on Postfix, a UNIX program responsible for sending emails. Because of this, during the installation process a pink screen will show up asking for Postfix configuration settings. Choose *Internet Site* to continue with the installation process and on the next screen, where it asks for the name of the system to send emails, type "monitor.devopsonlinestore.com".

The next screen will prompt for Nagios configuration, and you need to choose a password for the admin user "nagiosadmin" to be able to access the web interface. We will use the password "secret". After these configurations, the installation process will complete and you can test if the installation has worked by opening your web browser and visiting the URL http://nagiosadmin:secret@192.168.33.14/nagios3. If you have chosen another user or password, modify the URL using the correct credentials. If all goes well, you will see the Nagios administration web interface. Clicking the *Services* link will take you to a screen similar to one shown in figure 3.2.

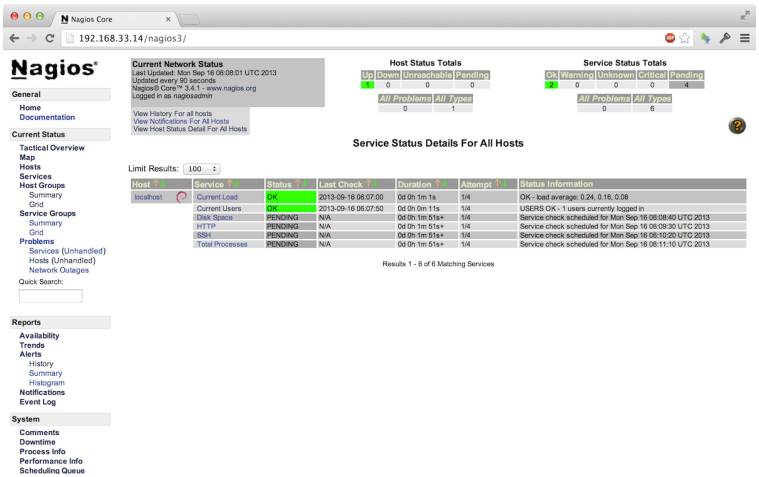

Fig. 3.2: Nagios administration interface

As you can see, some default checks – or services, in Nagios terminology – are now already preconfigured. For now, the server is only monitoring itself, acting as both supervisor and supervised. In this screen, each monitored server – or *host* – has a set of services. The Nagios server is represented by the name *localhost* and each row in the table represents an independent check. The green "OK" status means that the check has passed, while the status "PENDING" means that check has not yet executed. The default checks are:

- **Current load**: the load of a UNIX system measures how busy the CPU is. To be more precise, the load is a moving average of the CPU queue length for processes waiting to run. When the CPU begins to get overload, processes have to wait longer to run, increasing the queue and making the overall system run slower. This check sends an alert when the current load exceeds a certain limit.

- **Current users**: checks how many users are logged into the system. This is a simple way to detect server intrusion and to send alerts when the number of logged in users goes beyond the expected threshold.

- **Disk Space**: when a server is out of disk space, many problems start to show up. One of the most common reasons for this to happen is log generation on production systems. To avoid this problem it is important to monitor how much free space is still available. This check sends alerts when the free space reaches a certain limit.
- **HTTP**: checks that the server is accepting HTTP connections. In this case, Nagios has installed a web server that we are using to access its administration interface shown in figure 3.2. By default, this check will send alerts if it is unable to open an HTTP connection on port 80.
- **SSH**: checks that the server is accepting SSH connections on port 22 and send alerts when it finds a problem.
- **Total number of processes**: another way to assess whether the server is overloaded is by looking at the total number of processes. If many processes are running at the same time, the operating system will spend more time managing which one gets the CPU, leaving no time for any of them run effectively. This check sends alerts when the total number of processes exceeds a certain limit.

At first glance, it may seem that there are too many checks just to monitor a single host. However, following the same principle of writing a few assertions per test when developing code with TDD, more granular checks help you to identify the point of failure faster when a problem happens. Instead of receiving a generic warning that the server is in trouble, you will have specific information to identify the cause of the problem faster.

In the next sections we will discuss a little more about some Nagios concepts, configurations and learn how to supervise other hosts, such as adding more checks and setting it up to send alerts when something goes wrong.

3.2 Monitoring other hosts

With Nagios installed and running checks, we need to configure it to monitor our other production servers. To do this, we need to declare where the other servers are and what checks should be run against them. We will create a new file to define all configurations related to the online store:

```
vagrant@monitor$ sudo nano /etc/nagios3/conf.d/online_store.cfg
```

In this file we will declare the online store servers, adding new hosts with their respective IP addresses:

```
define host {
  use         generic-host
  host_name   192.168.33.10
  hostgroups  ssh-servers, debian-servers
}

define host {
  use         generic-host
  host_name   192.168.33.12
  hostgroups  ssh-servers, debian-servers
}
```

Whenever we change a configuration file, we need to reload Nagios, so it will detect the changes. For this, we use the command `reload`:

```
vagrant@monitor$ sudo service nagios3 reload
```

In Nagios, a *host* represents a device to be monitored, such as a server, a router or a printer. We may also group similar hosts in *host groups* and configure multiple servers at once to avoid duplication. The same host may belong to multiple host groups.

In the `/etc/nagios3/conf.d/online_store.cfg` configuration file, we define that our servers are members of two groups: `ssh-servers` and `debian-servers`. The `ssh-servers` group will add a check for the SSH service. The `debian-servers` group does not define any extra checks, but adds information such as the Debian logo that shows up as an icon in the Nagios web interface.

To make sure that the settings were applied successfully, you can reload the services page and wait until the new SSH checks are OK, as shown in figure 3.3.

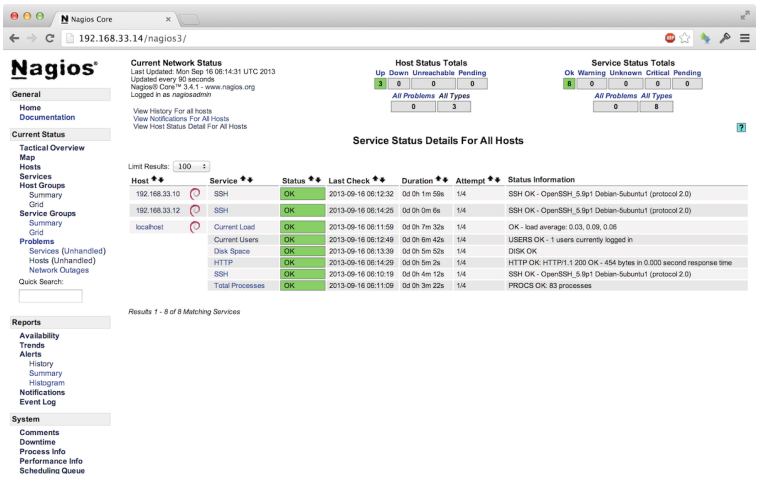

Fig. 3.3: Monitoring SSH service in more hosts

To simplify our configuration later, we will add our servers to two new host groups, one group called `web-servers` and the other called `db-servers`. We will do this by editing the same configuration file:

```
vagrant@monitor$ sudo nano /etc/nagios3/conf.d/online_store.cfg
```

You need to add the new host groups and change the configuration for the existing hosts to add them to the correct groups:

```
define hostgroup {
  hostgroup_name  db-servers
  alias           Database Servers
}

define hostgroup {
  hostgroup_name  web-servers
  alias           Web Servers
}

define host {
  use         generic-host
  host_name   192.168.33.10
```

```
  hostgroups  ssh-servers, debian-servers, db-servers
}

define host {
  use         generic-host
  host_name   192.168.33.12
  hostgroups  ssh-servers, debian-servers, web-servers
}
```

After reloading the configuration file – by running the same `sudo service nagios3 reload` command – you may check that there are new groups by clicking the *Host Groups* link on the Nagios administration screen.

Executing several checks and sending alerts when something is wrong are the most important features of a monitoring system like Nagios. In the next section, we will learn about the various types of checks available that we can use to verify if the online store is working properly.

3.3 Exploring Nagios service checks

Each Nagios service is associated with a command that runs on each verification. The command is nothing more than a script that can be run directly from the command line. Several commands are already installed in Nagios by default, but you can create your own scripts or install community plugins to extend the Nagios library.

Let's explore some of the pre-installed commands that are available in the `/usr/lib/nagios/plugins` directory. For example, the `check_ssh` command, which is executed by the SSH service:

```
vagrant@monitor$ cd /usr/lib/nagios/plugins
vagrant@monitor$ ./check_ssh 192.168.33.12
SSH OK - OpenSSH_5.9p1 Debian-5ubuntu1 (protocol 2.0)
vagrant@monitor$ echo $?
0
vagrant@monitor$ ./check_ssh 192.168.33.11
No route to host
vagrant@monitor$ echo $?
2
```

Note that when you run the `check_ssh%` command passing the web server IP address, it prints a one-line summary message with details about what happened. Usually this message has the format **<CHECK> <STATUS> - <DETAILED MESSAGE>**, but what determines the result of the check is not the message itself but the exit code of the command, represented by the `$?` variable in bash:

- **0 - OK**: indicates that the check executed successfully. Represented by the green status on the web interface.
- **1 - WARNING**: indicates that the check crossed the first warning threshold. Represented by the yellow status on the web interface.
- **2 - CRITICAL**: indicates that the check has failed and that something is wrong. Represented by the red status on the web interface.
- **3 - UNKNOWN**: indicates that the check could not decide if the service is doing well or not. Represented by the orange status on the web interface.

Another check we run is to verify if we can access the web server via HTTP:

```
vagrant@monitor$ ./check_http -H 192.168.33.12
Connection refused
HTTP CRITICAL - Unable to open TCP socket
vagrant@monitor$ echo $?
2
vagrant@monitor$ ./check_http -H 192.168.33.12 -p 8080
HTTP OK: HTTP/1.1 200 OK - 2134 bytes in 0.007 second ...
vagrant@monitor$ echo $?
0
```

Note that we need to pass the `-H` option to define which host should be checked. Also, since we have not specified the port to be checked, the `check_http` command uses port 80 by default. When we pass the `-p` option with the value 8080 – the port in which Tomcat is listening – the check passes successfully.

A last command for us to explore is checking how much disk space is left:

```
vagrant@monitor$ ./check_disk -H 192.168.33.12 -w 10% -c 5%
/usr/lib/nagios/plugins/check_disk: invalid option -- 'H'
Unknown argument
Usage:
 check_disk -w limit -c limit [-W limit] [-K limit] ...
vagrant@monitor$ ./check_disk -w 10% -c 5%
DISK OK - free space: / 74473 MB (97% inode=99%); ...
```

Note that, unlike the `check_http` command, the `check_disk` command does not accept the `-H` option and you can only check how much disk space is available on the local host. That happens because the HTTP service is a network service that can be accessed remotely. On the other hand, the `check_disk` command needs local server information to decide how much disk space is left and can not be run remotely.

Nagios has three options when you need to perform a check remotely for a service that is not accessible through the network:

- **Check by SSH**: using the `check_by_ssh` command, Nagios can open an SSH connection to the remote host and execute the command there. This is a simple option because it does not require installing any agent in the supervised host. However, with an increase in the number of remote checks, it may also increase the load on the supervisor because it will need to manage opening and closing several SSH connections.
- **Active checks with NRPE**: the *Nagios Remote Plugin Executor* or NRPE, is a plugin that allows you to execute checks that rely on local resources on a supervised host. An NRPE agent process runs on the supervised host and waits for the supervisor's check requests. When Nagios needs to execute a check, it will request it to the NRPE agent and collect the results when the execution completes. This type of verification is active because the command execution is requested by the supervisor host.
- **Passive checking with NSCA**: the *Nagios Service Check Acceptor* or NSCA, is a plugin that allows you to execute remote passive checks in the supervised host. An NSCA agent process runs on the supervised host and executes the checks periodically, sending the results to

an NSCA server running in the supervisor. This type of verification is passive because the command execution is started by the supervised host itself.

Figure 3.4 shows the different ways to execute checks with Nagios. Checks for network services, such as `check_http`, run directly from the supervisor host. Checks that need to run on the supervised host, may be actively executed with `check_by_ssh` or using the NRPE plugin. Finally, passive checks can run independently in the supervised host and the results are sent to Nagios via the NSCA plugin.

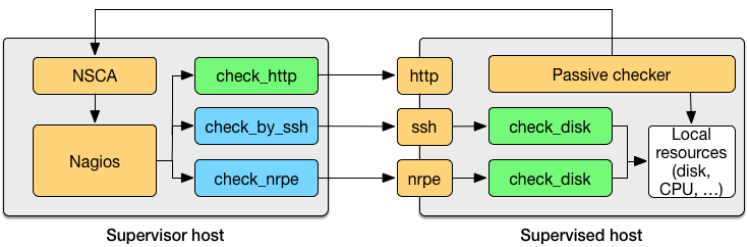

Fig. 3.4: Remote checks with Nagios

Now that you know the different types of Nagios checks and how they can be executed, let's make the online store monitoring infrastructure more robust.

3.4 Adding more specific checks

Besides the standard checks, you may declare your own services and associate them with a single host or host group. In our online store infrastructure, the database server is running MySQL as a network service. You can verify that this service is available by adding an associated check to the `db-servers` group at end of the `/etc/nagios3/conf.d/online_store.cfg` configuration file:

```
define hostgroup ...
define host ...
```

```
define service {
  service_description  MySQL
  use                  generic-service
  hostgroup_name       db-servers
  check_command        check_tcp!3306
}
```

After reloading Nagios and waiting for the next check run, a new "MySQL" service will appear with an OK status. This check uses the `check_tcp` command that tries to open a TCP connection on the specified port – in this case MySQL is running on port 3306. The Nagios syntax to pass arguments to a command is to separate them with an exclamation mark (`!`).

This check verifies that there is a process accepting TCP connections on port 3306, but it does not guarantee that such process is MySQL. To make sure that the database is actually there, we have to use the `check_mysql` command, which is already installed in the plugins directory `/usr/lib/nagios/plugins`. Running it directly from the command line, we can see which parameters it accepts:

```
vagrant@monitor$ /usr/lib/nagios/plugins/check_mysql --help
...
Usage:
  check_mysql [-d database] [-H host] [-P port] [-s socket]
        [-u user] [-p password] [-S]
...
```

So let's add a new check to the online store configuration file (remember that the backslash indicates that the command should be specified on the same line):

```
define hostgroup ...
define host ...
define service ...

define service {
  service_description  MySQL-lojavirtual
  use                  generic-service
```

```
  hostgroup_name       db-servers
  check_command        check_mysql_database!store \
                                            !storesecret \
                                            !store_schema
}
```

You may have noticed that the program we executed on the command line was `check_mysql` but the command you use in the service declaration is `check_mysql_database`. That is because the command must be defined in Nagios before it can be used by a service. In this case, the plugin has already created the commands, which we can find in the file `/etc/nagios-plugins/config/mysql.cfg`:

```
vagrant@monitor$ cat /etc/nagios-plugins/config/mysql.cfg
define command{
  command_name  check_mysql
  command_line  /usr/lib/nagios/plugins/check_mysql \
                                        -H '$HOSTADDRESS$'
}

...

define command{
  command_name  check_mysql_database
  command_line  /usr/lib/nagios/plugins/check_mysql \
                                        -d '$ARG3$' -H ...
}

...
```

The last command allows us to pass arguments for the username, password, and database schema to be verified. We can also see the order in which the parameters should be passed when we configure our service: Nagios will replace the tokens `$ARG1$`, `$ARG2$`, and `$ARG3$` with the values we define in our service, split by the exclamation mark.

Now that we are monitoring our database server, it is time to add some checks to the web server. Following the same reasoning as before, the network

services exposed by the web server are Tomcat HTTP and HTTPS connections. The command that we can use to perform such checks is `check_http`, also installed in the plugins directory `/usr/lib/nagios/plugins`. Executing it directly from the command line, we can see which arguments it accepts:

```
vagrant@monitor$ /usr/lib/nagios/plugins/check_http --help
...
Usage:
 check_http -H <vhost> | -I <IP-address> [-u <uri>] [-p <port>]
  [-w <warn time>] [-c <critical time>] [-t <timeout>] [-L]
  [-a auth] [-b proxy_auth]
  [-f <ok|warning|critcal|follow|sticky|stickyport>]
  [-e <expect>] [-s string] [-l]
  [-r <regex> | -R <ignorecase regex>]
  [-P string] [-m <min_pg_size>:<max_pg_size>] [-4|-6] [-N]
  [-M <age>] [-A string] [-k string] [-S <version>] [--sni]
  [-C <warn_age>[,<crit_age>]] [-T <content-type>] [-j method]
NOTE: One or both of -H and -I must be specified
...
```

In this case, the plugin already declares Nagios commands in the file `/etc/nagios-plugins/config/http.cfg`, but none of them accepts the argument `-p` to specify the port. Since Tomcat is not running on the default HTTP port (80), we need to declare our own command before associating it with a new service for the `web-servers` group. We will put that in the online store configuration file `/etc/nagios3/conf.d/online_store.cfg`:

```
define hostgroup ...
define host ...
define service ...
define service ...

define command {
  command_name  check_tomcat_http
  command_line  /usr/lib/nagios/plugins/check_http \
                                          -H '$HOSTADDRESS$' \
```

```
                                        -p '$ARG1$' \
                                        -u '$ARG2$' \
                                        -e 'HTTP/1.1 200 OK'
}

define service {
  service_description Tomcat
  use                 generic-service
  hostgroup_name      web-servers
  check_command       check_tomcat_http!8080 \
                                       !'/devopsnapratica/'
}
```

The argument `-H` represents the supervised host address, the first argument represents the HTTP port and the second argument is the URI. Our command also sets the `-e` option to ensure that the server response is successful and returns a status of: `HTTP/1.1 200 OK`.

Finally, let's add one last check to guarantee that the SSL connection also works on port 8443:

```
define hostgroup ...
define host ...
define service ...
define service ...
define command ...
define service ...

define command {
  command_name  check_tomcat_https
  command_line  /usr/lib/nagios/plugins/check_http \
                                        -H '$HOSTADDRESS$' \
                                        --ssl=3 \
                                        -p '$ARG1$' \
                                        -u '$ARG2$' \
                                        -e 'HTTP/1.1 200 OK'
}

define service {
  service_description Tomcat SSL
```

```
  use                  generic-service
  hostgroup_name       web-servers
  check_command        check_tomcat_https!8443 \
                                 !'/devopsnapratica/admin/'
}
```

After reloading Nagios and waiting for the new checks to execute, the services page in the administration screen should look like picture 3.5.

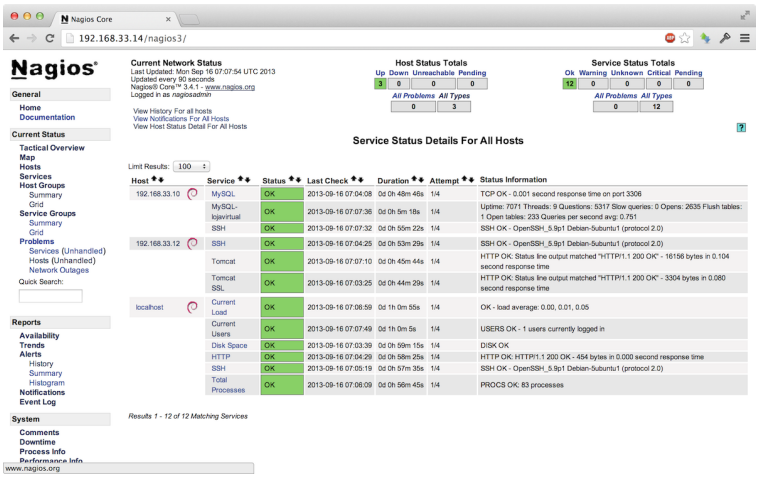

Fig. 3.5: All online store checks are passing

By the end of this section, the complete configuration file `/etc/nagios3/conf.d/online_store.cfg` will look like:

```
define hostgroup {
  hostgroup_name  db-servers
  alias           Database Servers
}

define hostgroup {
  hostgroup_name  web-servers
  alias           Web Servers
}
```

```
define host {
  use             generic-host
  host_name       192.168.33.10
  hostgroups      ssh-servers, debian-servers, db-servers
}

define host {
  use             generic-host
  host_name       192.168.33.12
  hostgroups      ssh-servers, debian-servers, web-servers
}

define service {
  service_description  MySQL
  use                  generic-service
  hostgroup_name       db-servers
  check_command        check_tcp!3306
}

define service {
  service_description  MySQL-lojavirtual
  use                  generic-service
  hostgroup_name       db-servers
  check_command        check_mysql_database!store \
                                            !storesecret \
                                            !store_schema
}

define command {
  command_name  check_tomcat_http
  command_line  /usr/lib/nagios/plugins/check_http \
                                        -H '$HOSTADDRESS$' \
                                        -p '$ARG1$' \
                                        -u '$ARG2$' \
                                        -e 'HTTP/1.1 200 OK'
}

define service {
  service_description Tomcat
```

```
  use                 generic-service
  hostgroup_name      web-servers
  check_command       check_tomcat_http!8080 \
                                  !'/devopsnapratica/'
}

define command {
  command_name  check_tomcat_https
  command_line  /usr/lib/nagios/plugins/check_http \
                                    -H '$HOSTADDRESS$' \
                                    --ssl=3 \
                                    -p '$ARG1$' \
                                    -u '$ARG2$' \
                                    -e 'HTTP/1.1 200 OK'
}

define service {
  service_description Tomcat SSL
  use                 generic-service
  hostgroup_name      web-servers
  check_command       check_tomcat_https!8443 \
                              !'/devopsnapratica/admin/'
}
```

Now that we have more detailed checks about the online store operation, the last thing we need to do to have a robust monitoring system is to setup and receive alerts.

3.5 Receiving alerts

Having detailed checks of your system is only the first step to build a robust monitoring system. It will be worthless if your monitoring system detects a problem and you are not warned. Therefore, it is important to configure alerts to receive information about problems as soon as possible.

There are several Nagios plugins to send alerts in various ways: email, pager, SMS, instant messaging such as MSN, ICQ, Yahoo, or anything you can do in the command line. You may even write your own custom command to define how Nagios will send alerts. By default, Nagios is pre-configured with

two commands to send email alerts: one notifies that the host is in trouble and the other notifies that some service is in trouble. We can see the definition of these commands in the `/etc/nagios3/commands.cfg` file:

```
vagrant@monitor$ cat /etc/nagios3/commands.cfg
...
define command {
  command_name  notify-host-by-email
  command_line  /usr/bin/printf "%b" ... | /usr/bin/mail -s ...
}

define command {
  command_name  notify-service-by-email
  command_line  /usr/bin/printf "%b" ... | /usr/bin/mail -s ...
}
...
```

Besides these commands, there are two other Nagios objects that can be configured to specify who receives the alerts and when: **contacts** and **time periods**. Time periods specify when alerts can be sent and are defined in the `/etc/nagios3/conf.d/timeperiods_nagios2.cfg` file. Looking at the contents of the file we can see the different options, including the famous 24x7 support period – 24 hours a day, 7 days a week:

```
vagrant@monitor$ cat /etc/nagios3/conf.d/timeperiods_nagios2.cfg
...
define timeperiod {
  timeperiod_name 24x7
  alias           24 Hours A Day, 7 Days A Week
  sunday          00:00-24:00
  monday          00:00-24:00
  tuesday         00:00-24:00
  wednesday       00:00-24:00
  thursday        00:00-24:00
  friday          00:00-24:00
  saturday        00:00-24:00
}
...
```

Contacts are the people who receive alerts. In our case, we use the contact defined by Nagios for the `root` user in the file `/etc/nagios3/conf.d/contacts_nagios2.cfg`:

```
vagrant@monitor$ cat /etc/nagios3/conf.d/contacts_nagios2.cfg
...
define contact {
  contact_name                    root
  alias                           Root
  service_notification_period     24x7
  host_notification_period        24x7
  service_notification_options    w,u,c,r
  host_notification_options       d,r
  service_notification_commands   notify-service-by-email
  host_notification_commands      notify-host-by-email
  email                           root@localhost
}
...
```

In the contact definition you choose which period the contact will be on-call, its email address, the commands executed to send alerts and the notification options. In this case, we will be notified whenever the host is down (`d`) or recovered (`r`) and when the services go into a warning (`w`), unknown (`u`), critical (`c`) or recovered (`r`) state. Nagios also supports **contact groups** when you want to alert a whole team rather than a single person, but we will not use this feature.

To ensure that our infrastructure can send valid alerts, we must choose which messaging service Nagios will use. Although nobody likes to be woken up at night, operations teams need 24x7 support for their systems and therefore they prefer to be notified by pager or SMS. However, it is difficult to find a free service for sending SMS. Some options for paid online services with international SMS coverage are **Twilio** (http://www.twilio.com/) and **Nexmo** (http://www.nexmo.com/) .

Another widely used paid service is **PagerDuty** (http://www.pagerduty.com/) , that allows you to manage groups, contacts, on-call lists and supports different ways to send alerts in a single place, independent of Nagios. PagerDuty supports SMS, email, or even iPhone or Android push notifica-

tions. Before you start paying for the monthly plan, you can start with a 30 days a trial and PagerDuty has a very detailed documentation explaining how to integrate it with Nagios.

In our case, to avoid a dependency on an external paid system, we will use the standard Nagios configuration to send alerts by email. The only configuration required is to create a contact in Nagios to receive the online store alerts. We will do it by editing the contacts configuration file `/etc/nagios3/conf.d/contacts_nagios2.cfg`, replacing the email address `root@localhost` with your true email address:

```
...
define contact {
  contact_name  root
  ...
  email         <your_email>@<your_provider>.com
}
...
```

For this setting to take place, we need to reload the Nagios service for one last time:

```
vagrant@monitor$ sudo service nagios3 reload
 * Reloading nagios3 monitoring daemon configuration files
 nagios3 ...done.
```

From now on, any alerts will be sent directly to your email. To test if the alert is working properly, you can stop the SSH service and wait for execution of the next Nagios check:

```
vagrant@monitor$ sudo service ssh stop
ssh stop/waiting
```

Nagios will attempt to execute the check four times before declaring the service unavailable. Once the service appears with a red status in the Nagios administration screen and the number of attempts reach "4/4", you should receive an alert by email. To restore everything back to normal, start the SSH service:

```
vagrant@monitor$ sudo service ssh start
ssh start/running, process 3424
```

Once Nagios detects that the service is back to normal, you will receive another email notifying you that the service has been restored.

3.6 A problem hits production, now what?

Now that we have configured a monitoring system to run checks and send alerts when the online store infrastructure has problems, we will simulate a failure scenario of a server going down. This is also a good way to test if the monitoring system is working properly.

Let's imagine that our database server suffered a hardware problem and stopped working. To simulate this scenario, logout of the monitor server by running the `logout` command or simply typing `CTRL-D` and ask Vagrant to destroy the database server:

```
vagrant@monitor$ logout
Connection to 127.0.0.1 closed.
$ vagrant destroy db
    db: Are you sure you want to destroy the 'db' VM? [y/N] y
==> db: Forcing shutdown of VM...
==> db: Destroying VM and associated drives...
```

Within a few minutes, the monitoring system will detect that the server is unavailable and will send an alert. From this moment on, users can no longer access the online store to place orders. Being the engineer responsible for the operation in production, you receive the alert and answer promptly to confirm the problem and now you need to restore the service.

Looking at the Nagios console, you will see something similar to figure 3.6. The database host and all its services are red, but the HTTP service check on the web server also fails, indicating that the database problem is also affecting the online store.

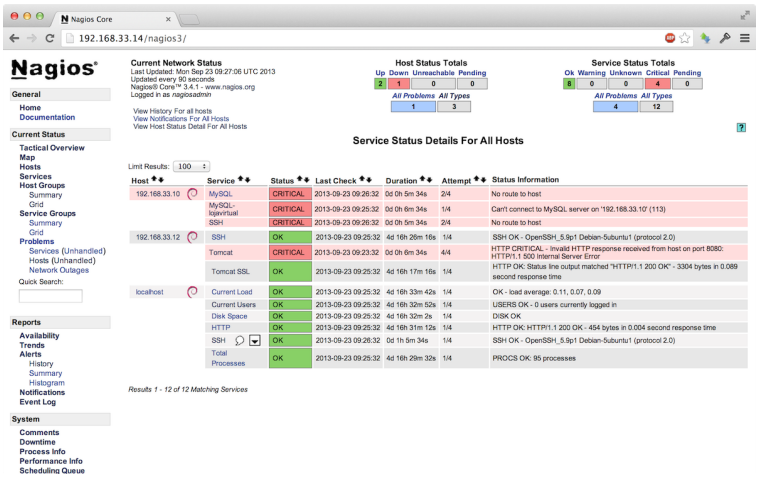

Fig. 3.6: Checks failing when a problem hits production

To restore everything back to normal, you will need to re-install the database server. However, we did it manually in chapter 2. Repeating all those steps manually will take a long time and, in moments of crisis like this, time is a scarce resource. There must be a better way to install and configure these servers.

Chapter 4

Infrastructure as code

System administrators learn the power of the command line early in their careers and spend most of their day in the terminal, controlling their company's infrastructure. It is often said that they are changing the **system configuration,** whether installing new servers, installing new software, upgrading existing software, editing configuration files, restarting processes, reading log files, making and restoring backups, creating new user accounts, managing permissions, helping developers to debug problems or writing scripts to automate repeatable tasks.

Each of these tasks changes the state of the system slightly and can potentially cause problems if the change is not performed correctly. That is why many system administrators do not give developers access to production servers: to avoid undesirable changes, and due to lack of trust. However, in many companies, these tasks are performed manually on production servers by the system administrators themselves, who can also make mistakes.

One of the main principles of DevOps is to invest in automation. Automation allows you to perform tasks quicker and reduces the chance of human error. An automated process is more reliable and can be more easily audited. As the number of servers increases, automated processes become essential. Some system administrators write their own scripts to automate the most standard tasks. The problem is that each one writes their own version of these scripts and it is difficult to reuse them in other contexts. On the other hand, developers are usually good at writing modularized and reusable code.

With the growth of the DevOps culture and the increased collaboration between system administrators and developers, several tools have emerged to try to standardize the automated management of infrastructure configuration. Such tools allow infrastructure to be treated as code: using version control, writing tests, packaging and distributing common modules and obviously performing the configuration changes on the server.

This practice is known as **infrastructure as code** and will allow us to solve the problem we found at the end of chapter 3. Instead of reinstalling and reconfiguring everything manually, we will use a configuration management tool to automate the provisioning, configuration and deployment of our online store application.

4.1 Provision, configure or deploy?

When buying a laptop or a new mobile phone, all you have to do is to switch it on and it's ready to be used. Instead of sending individual components and a manual explaning how to assemble your machine, the manufacturer has already done all the work required for you to enjoy your new machine. As a user, you just have to install programs and restore your files to consider the machine yours. This preparation process for the end user is known as **provisioning.**

The term **provisioning** is commonly used by telecommunications companies and operations teams to refer to the initial configuration stages of preparing a new resource: provisioning a mobile device, provisioning your Internet access, provisioning a server, provisioning a new user account, and so on. Provisioning a mobile device involves, among other things: reserving

a new line, configuring the network equipment that allows you to complete your calls, configuring extra services such as SMS or email, and finally associating everything with the chip that's installed in your mobile phone.

For servers, the provisioning process varies from company to company depending on its infrastructure and the separation of responsibilities between the teams. If the company has its own infrastructure, the provisioning process involves the purchase and physical installation of the new server in the datacenter. If the company has a virtualized infrastructure, the provisioning process only requires adding a new virtual machine to the server. Likewise, if the operations team considers the development team as their "end user," the provisioning process finishes when the server is accessible on the network, even if the application itself is still not running. If we consider the actual users as the "end users," then the provisioning process only ends when the server and the application are running and accessible on the network.

To avoid further confusion throughout the book, it is worth our looking at the end-to-end process and to define an appropriate terminology. Imagining the longest scenario of a company that owns its infrastructure but has no extra capacity to use, the required steps to make a new application live would be:

1) **Buying hardware**: In large companies, this step begins a procurement process that includes several justification and approval rounds, since investing money in hardware influences the company's accounting and financial planning.

2) **Physical hardware installation**: This step encompasses mounting the new server in a rack in the datacenter, as well as installing power and network cables, etc.

3) **Installing and configuring the operating system**: Once the server is connected, you have to install an operating system and configure basic hardware items such as: network interfaces, storage (disk, partitions, network volumes), setup user authentication and authorization, managing the root account password, configuring packages repository, etc.

4) **Installing and configuring common services**: Beyond the operating system configuration, there are basic infrastructure services that will probably

be common among all servers, such as: DNS, NTP, SSH, log collection and rotation, backups, firewall, printing, etc.

5) **Installing and configuring the application**: Finally, you must install and configure everything that will make this server unique: middleware components, the application itself, as well as their configurations.

Throughout the book, we will use the term **provisioning** to refer to the process that involves steps 1 - 4, which includes all the required activities to make a server useful, regardless of the reason why it was requested. We will use the term **deploy** when referring to step 5. Although deployment requires a provisioned server, the server's lifecycle is different from the application's lifecycle. The same server may be used by multiple applications, and each application can have tens or even hundreds of deploys while the provisioning of new servers occurs much less often.

In the worst case, the provisioning process can take weeks or even months depending on the company's processes. For this reason, it is common to see development teams defining hardware requirements for their production environments fairly in advance, and starting this process with the operations team in the early stages of the project, so as to avoid delays when the application is ready to go live. Some collaboration may happen early in the process, but most of the time both teams work in parallel without knowing what the other one is doing.

Even worse, many of these critical architectural decisions about the number of servers, hardware configuration or operating system selection are made at the beginning of the project, in the moment where there is the least amount of information and knowledge about the real needs of the system.

One of the most important aspects of the DevOps culture is to recognize this conflict of interest and to create an environment of collaboration between development and operation teams. Sharing practices and tools allows the architecture to adapt and evolve as teams learn more about the system running in the real world: in production. This holistic view also helps to find solutions that simplify the process of buying and provisioning, removing stages or using automation to make them more efficient.

A good example of this process simplification is the use of technologies such as hardware virtualization and *cloud computing* or simply *cloud*. Top cloud providers allow you to have a new server in a matter of minutes, with the click of a button or a simple API call!

4.2 Configuration management tools

In the previous section, we explained the difference between provisioning and deployment. Except for steps 1 and 2, which involve the purchase and installation of physical hardware, the remaining three steps involve some form of system configuration: whether to the operating system itself, or the base installed software or the application that runs on top of it. Figure 4.1 shows the different lifecycle stages of a server and the scope of configuration management.

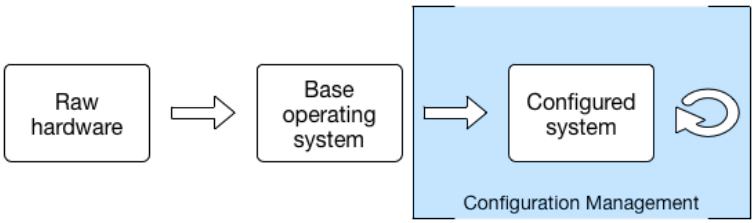

Fig. 4.1: Configuration management scope

There are several configuration management tools: **shell scripts** (the command line is also known as the *shell*), **Puppet** (http://www.puppetlabs.com/puppet/) , **Chef** (http://www.opscode.com/chef/) , **Ansible** (http://www.ansibleworks.com/tech/) , **Salt** (http://saltstack.com/community) , **Pallet** (http://palletops.com/) and **CFEngine** (http://cfengine.com/) are some of the most popular ones.

Most of these tools have a company behind their development that is responsible for growing the community and the ecosystem around them. All of these companies are contributing to the evolution of the DevOps community and, despite competing with each other, their biggest competitor is still the lack of tools usage. As unbelievable as it might seem, most companies still

manage their servers manually or have proprietary scripts written so long ago that nobody can maintain them anymore.

The big difference between proprietary shell scripts and other tools is that generally the script only installs and configures the server for the first time. It serves as an executable documentation of the steps performed on a first installation, but if the script is executed again, it probably won't work. The system administrator must be well disciplined and write extra code to deal with situations where some package is already installed or needs to be removed or updated.

The other tools have an important feature, known as **idempotency**: It allows you to execute the same code over and over again and they will only apply the required changes. If a package has been already installed, it will not be installed again. If a service is already running, it will not be restarted. If the configuration file already has the expected content, it will not be changed.

Idempotency allows you to write infrastructure code in a declarative way. Instead of an instruction saying "install package X" or "create user Y," you can say,"I want package X to be installed," or "I want user Y to exist." You declare the desired state and when the tool executes, if the package or the user already exist, nothing will happen.

The next thing you may be wondering is "Which of these tools should I use?" In this chapter, we will use Puppet because its language allows us to demonstrate this declarative property, as well as because it is a mature tool that has concepts familiar to system administrators and has been well adopted by the community. But the truth is that using any of these tools is better than nothing. The tools in this space are constantly evolving and it is important that you become familiar with a few of them before deciding which one is best for your situation.

[Introduction to Puppet: resources, providers, manifests and dependencies]

Each configuration management tool has its own terminology to refer to the elements of its language and the components of its ecosystem. At a fundamental level, each command that you declare in the language is a **directive** and you can define a set of them in a **directive file,** which is the equivalent of a source code file in a programming language like Java or Ruby.

In Puppet, these directives are called **resources.** A few examples of resources that you may declare in Puppet are: packages, files, users, groups, services, executable scripts, among others. The file in which you declare a set of directives is called a **manifest.** When writing Puppet code, you will spend most of the time creating and editing these manifest files.

To become familiar with the syntax of the Puppet language and its execution model, we will rebuild the database server VM and login via SSH on the newly created VM:

```
$ vagrant up db
Bringing machine 'db' up with 'virtualbox' provider...
==> db: Importing base box 'hashicorp/precise32'...
==> db: Matching MAC address for NAT networking...
==> db: Checking if box 'hashicorp/precise32' is up to date...
==> db: Setting the name of the VM: blank_db_1394854529922_29194
==> db: Clearing any previously set network interfaces...
==> db: Preparing network interfaces based on configuration...
    db: Adapter 1: nat
    db: Adapter 2: hostonly
==> db: Forwarding ports...
    db: 22 => 2222 (adapter 1)
==> db: Running 'pre-boot' VM customizations...
==> db: Booting VM...
==> db: Waiting for machine to boot. This may take a few
        minutes...
    db: SSH address: 127.0.0.1:2222
    db: SSH username: vagrant
    db: SSH auth method: private key
==> db: Machine booted and ready!
==> db: Configuring and enabling network interfaces...
==> db: Mounting shared folders...
    db: /vagrant => /private/tmp/blank
$ vagrant ssh db
Welcome to Ubuntu 12.04 LTS (GNU/Linux ...
vagrant@db$
```

Leveraging the fact that our Vagrant box already has Puppet installed, we can start using it by creating an empty manifest file called `db.pp`. By default, manifest files use the `.pp` extension:

```
vagrant@db$ nano db.pp
```

Let's start by declaring a resource for the `mysql-server` package and specify that we want it installed in the system. This is achieved by setting the `ensure` parameter to `installed` on the `package` resource:

```
package { "mysql-server":
  ensure => installed,
}
```

If you compare this code with the first command we executed in chapter 2 – `sudo apt-get install mysql-server` – you will notice that they are very similar. The main difference is the **declarative** nature of the Puppet language. In the manual command, we instruct the package manager to install the `mysql-server` package, while in Puppet you declare the system's desired end state.

Manifests are not a list of commands to be executed. When Puppet runs, it will compare the current system state with the desired state declared in the manifest. It will calculate the difference and perform only the necessary changes.

After saving the file, you can run the Puppet code with the following command:

```
vagrant@db$ sudo puppet apply db.pp
err: /Stage[main]//Package[mysql-server]/ensure: change from
purged to present failed: Execution of ...
...
E: Unable to fetch some archives, maybe run apt-get update or ...

notice: Finished catalog run in 6.62 seconds
```

Notice that Puppet tried to change the status of the `mysql-server` package from purged to present, however it encountered errors, because we forgot to run the `apt-get update` command, just like we did in chapter 2. To fix that, we will declare a new `exec` and set a dependency by saying that it must run before the `mysql-server` package resource. We will change the contents of the `db.pp` file to:

```
exec { "apt-update":
  command => "/usr/bin/apt-get update"
}

package { "mysql-server":
  ensure  => installed,
  require => Exec["apt-update"],
}
```

To understand this manifest file, we need to learn a few more features of the Puppet language. Every resource has a name – the quoted string after the resource declaration and before the ":" – which serves as a unique identifier. When a resource needs to refer to another resource, we use the syntax with the resource type's first letter uppercased and the resource name between square brackets.

Puppet does not guarantee that the execution order will respect the order in which resources are declared in the manifest file. When there is a dependency, you must declare it explicitly. In this case, we add the `require` parameter to the `package` resource to ensure that the `apt-get update` command will execute before the MySQL installation. Now Puppet will install the `mysql-server` package when we run it again:

```
vagrant@db$ sudo puppet apply db.pp
notice: /Stage[main]//Exec[apt-update]/returns: executed \
        successfully
notice: /Stage[main]//Package[mysql-server]/ensure: ensure \
        changed 'purged' to 'present'
notice: Finished catalog run in 59.98 seconds
```

You can see that Puppet changed the status of the `mysql-server` package from purged to present. To confirm that the package was actually installed, you can run the following command:

```
vagrant@db$ aptitude show mysql-server
Package: mysql-server
State: installed
...
```

Note that the package manager says that the current state of the `mysql-server` package is installed. To understand the declarative nature of the manifest, try running Puppet again:

```
vagrant@db$ sudo puppet apply db.pp
notice: /Stage[main]//Exec[apt-update]/returns: executed \
        successfully
notice: Finished catalog run in 6.41 seconds
```

This time, the `apt-update` resource was executed, but nothing happened with the `mysql-server` package, since it was already installed, satisfying what we declared in our Puppet manifest.

Another difference you may have noticed between the Puppet manifest and the manual command from chapter 2 is that we don't have to tell Puppet which package manager to use. That is because Puppet resources are abstract. There are several different implementations – called **providers** – for the various operating systems. For the `package` resource, Puppet has several providers: `apt`, `rpm`, `yum`, `gem`, among others.

At runtime, based on the system information, Puppet will choose the most suitable provider. To see a list of all the available providers available for a given resource, as well as detailed documentation about the parameters accepted by each resource, you can run the command:

```
vagrant@db$ puppet describe package

package
=======
Manage packages. ...
...
Providers
---------
    aix, appdmg, apple, apt, aptitude, aptrpm, blastwave,
    dpkg, fink, freebsd, gem, hpux, macports, msi, nim,
    openbsd, pacman, pip, pkg, pkgdmg, pkgutil, portage,
    ports, portupgrade, rpm, rug, sun, sunfreeware, up2date,
    urpmi, yum, zypper
```

Now that we understand the basic mechanics of Puppet, we will stop running it manually inside the server and start writing infrastructure code outside the VM. We will use Vagrant to help us provision the server.

Starting from scratch using Puppet with Vagrant

First of all, let's logout from the `db` server and destroy it again, as we did at the end of chapter 3:

```
vagrant@db$ logout
Connection to 127.0.0.1 closed.
$ vagrant destroy db
    db: Are you sure you want to destroy the 'db' VM? [y/N] y
==> db: Forcing shutdown of VM...
==> db: Destroying VM and associated drives...
```

In the same directory where you created your `Vagrantfile`, create a new directory called `manifests` and, inside it, create a new file called `db.pp` with the same content as the previous section. The new directory structure should look like this:

```
.
 Vagrantfile
 manifests
     db.pp
```

Now you have to edit the Vagrant configuration file – `Vagrantfile` – to setup Puppet as the provisioning tool for the `db` virtual machine:

```
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  config.vm.box = "hashicorp/precise32"

  config.vm.define :db do |db_config|
    db_config.vm.hostname = "db"
    db_config.vm.network :private_network,
                         :ip => "192.168.33.10"
    db_config.vm.provision "puppet" do |puppet|
```

```
      puppet.manifest_file = "db.pp"
    end
  end

  config.vm.define :web do |web_config|
    web_config.vm.hostname = "web"
    web_config.vm.network :private_network,
                          :ip => "192.168.33.12"
  end

  config.vm.define :monitor do |monitor_config|
    monitor_config.vm.hostname = "monitor"
    monitor_config.vm.network :private_network,
                              :ip => "192.168.33.14"
  end
end
```

With this change, as soon as we bring up a new database server VM, Vagrant will run Puppet automatically, without the need to login via SSH and execute the command manually. The output from running Puppet will be shown immediately after Vagrant's normal output and it will also report any errors. For example, with our initial implementation, it will show the `mysql-server` package getting installed:

```
$ vagrant up db
Bringing machine 'db' up with 'virtualbox' provider...
==> db: Importing base box 'hashicorp/precise32'...
==> db: Matching MAC address for NAT networking...
==> db: Checking if box 'hashicorp/precise32' is up to date...
==> db: Setting the name of the VM: blank_db_1394854957677_21160
==> db: Clearing any previously set network interfaces...
==> db: Preparing network interfaces based on configuration...
    db: Adapter 1: nat
    db: Adapter 2: hostonly
==> db: Forwarding ports...
    db: 22 => 2222 (adapter 1)
==> db: Running 'pre-boot' VM customizations...
==> db: Booting VM...
==> db: Waiting for machine to boot. This may take a few \
```

```
        minutes...
    db: SSH address: 127.0.0.1:2222
    db: SSH username: vagrant
    db: SSH auth method: private key
==> db: Machine booted and ready!
==> db: Configuring and enabling network interfaces...
==> db: Mounting shared folders...
    db: /vagrant => /private/tmp/blank
    db: /tmp/vagrant-puppet-2/manifests => \
        /private/tmp/blank/manifests
==> db: Running provisioner: puppet...
Running Puppet with db.pp...
stdin: is not a tty
notice: /Stage[main]//Exec[apt-update]/returns: executed \
        successfully
notice: /Stage[main]//Package[mysql-server]/ensure: ensure \
        changed 'purged' to 'present'
notice: Finished catalog run in 71.68 seconds
```

Now that we know how to use Vagrant and Puppet to provision our server, it is time to fully recover the online store's production environment.

4.3 Provisioning the database server

We have a simple manifest that performs the first step of the database server installation process. If you recall from chapter 2, the next step we took was to create the configuration file `/etc/mysql/conf.d/allow_external.cnf` to allow remote access to MySQL server. We will do that with Puppet using the `file` resource, which declares that a given file should exist in the configured system. We change the contents of our `db.pp` file to:

```
exec { "apt-update":
  command => "/usr/bin/apt-get update"
}

package { "mysql-server":
  ensure  => installed,
```

```
  require => Exec["apt-update"],
}

file { "/etc/mysql/conf.d/allow_external.cnf":
  owner   => mysql,
  group   => mysql,
  mode    => 0644,
  content => "[mysqld]\n  bind-address = 0.0.0.0",
  require => Package["mysql-server"],
}
```

By default, the file resource name represents the full path where the file will be created in the system and the `content` parameter has its content. To specify a different path, you can also use the `path` parameter. The %owner , group `and` modefile and its permissions.

To apply the new configuration, instead of destroying and starting a new VM, you can use Vagrant's `provision` command to simply run Puppet in an already running VM:

```
$ vagrant provision db
  ==> db: Running provisioner: puppet...
Running Puppet with db.pp...
stdin: is not a tty
notice: /Stage[main]//Exec[apt-update]/returns: executed \
        successfully
notice: /Stage[main]//File[ /etc/mysql/conf.d \
                            /allow_external.cnf]/ \
        ensure: defined content as \
        '{md5}4149205484cca052a3bfddc8ae60a71e'
notice: Finished catalog run in 4.36 seconds
```

This time Puppet created the file `/etc/mysql/conf.d/allow_external.cnf`. To determine when a file needs to be changed, Puppet calculates an MD5 *checksum* of its content. The MD5 *checksum* is a hash algorithm widely used to verify the integrity of a file. A small change in the file's content makes its MD5 *checksum* totally different.

In our case, the contents of the configuration file is small and we can declare it entirely in the manifest. However, it is common to have configuration

files with tens or hundreds of lines. Managing long files as a big string in Puppet code is not a good idea. We can use **templates** to define a file's content. Puppet supports Ruby's ERB templates, a templating system that allows you to embed Ruby code within a text file.

To use a template, we first need to extract the contents to a new file `allow_ext.cnf`, which must exist along with the manifest file:

```
.
 Vagrantfile
 manifests
     allow_ext.cnf
     db.pp
```

Change the contents of the `allow_ext.cnf` file slightly, so that Puppet will apply the change to the file on the next run:

```
[mysqld]
   bind-address = 9.9.9.9
```

Finally, we switch the `content` parameter of the `file` resource to use the new template:

```
exec ...
package ...

file { "/etc/mysql/conf.d/allow_external.cnf":
  owner   => mysql,
  group   => mysql,
  mode    => 0644,
  content => template("/vagrant/manifests/allow_ext.cnf"),
  require => Package["mysql-server"],
}
```

When running Puppet again, you will see that the MD5 *checksum* will change and that the file contents will be updated:

```
$ vagrant provision db
==> db: Running provisioner: puppet...
Running Puppet with db.pp...
```

```
stdin: is not a tty
notice: /Stage[main]//Exec[apt-update]/returns: executed \
        successfully
notice: /Stage[main]//File[ /etc/mysql/conf.d \
                            /allow_external.cnf]/ \
        content: content changed \
                 '{md5}4149205484cca052a3bfddc8ae60a71e' to \
                 '{md5}8e2381895fcf40fa7692e38ab86c7192'
notice: Finished catalog run in 4.60 seconds
```

Now we need to restart the service, so that MySQL will notice the configuration change. We can do that by declaring a new `service` resource:

```
exec ...
package ...
file ...

service { "mysql":
  ensure     => running,
  enable     => true,
  hasstatus  => true,
  hasrestart => true,
  require    => Package["mysql-server"],
}
```

The service has new parameters: `ensure => running`, which ensure that the service is running; `enable`, which ensures that the service runs whenever the server is restarted; `hasstatus` and %hasrestart , which informs Puppet that the service supports the status and restart commands; and finally, the require parameter, which declares a dependency on the Package["mysql-server"]

If you try to run Puppet, you will see that nothing happens because the service has already been started when the package was installed. To restart the service every time the configuration file changes, we need to declare a new type of dependency on the resource `File["/etc/mysql/conf.d/allow_external.cnf"]`:

```
exec ...
package ...
```

```
file { "/etc/mysql/conf.d/allow_external.cnf":
  owner   => mysql,
  group   => mysql,
  mode    => 0644,
  content => template("/vagrant/manifests/allow_ext.cnf"),
  require => Package["mysql-server"],
  notify  => Service["mysql"],
}

service ...
```

The `notify` parameter defines an execution dependency: whenever the `file` resource is changed, it will notify the `service` resource to execute. This time, if we change the contents of our `allow_ext.cnf` template back to `0.0.0.0` and execute Puppet, the service will be restarted:

```
$ vagrant provision db
==> db: Running provisioner: puppet...
Running Puppet with db.pp...
stdin: is not a tty
notice: /Stage[main]//Exec[apt-update]/returns: executed \
        successfully
notice: /Stage[main]//File[ /etc/mysql/conf.d \
                            /allow_external.cnf]/ \
        content: content changed \
                 '{md5}8e2381895fcf40fa7692e38ab86c7192' to \
                 '{md5}907c91cba5e1c7770fd430182e42c437'
notice: /Stage[main]//Service[mysql]: Triggered 'refresh' \
        from 1 events
notice: Finished catalog run in 6.88 seconds
```

With this change, the database is running again and one of our Nagios checks should go green. To fix the next check, we must complete the database configuration by creating the online store's schema and user. We will create the schema by declaring another `exec` resource with the same command used in chapter 2:

```
exec ...
package ...
```

```
file ...
service ...

exec { "store-schema":
  unless  => "mysql -uroot store_schema",
  command => "mysqladmin -uroot create store_schema",
  path    => "/usr/bin/",
  require => Service["mysql"],
}
```

Besides the dependency with the `Service["mysql"]` resource, we are using a new `unless` parameter, which specifies a test command: If its exit code is zero, the main command will not be executed. This is how you make an `exec` resource idempotent. Unlike the `exec` we are using to run the `apt-get update` command, in this case it is important that Puppet does not try to create a new schema every time it runs.

Like we did in chapter 2, we also need to revoke the anonymous account access to MySQL before creating the user to access our new schema. We will do that by declaring a new `exec` resource:

```
exec ...
package ...
file ...
service ...
exec ...

exec { "remove-anonymous-user":
  command => "mysql -uroot -e \"DELETE FROM mysql.user \
                                WHERE user=''; \
                                FLUSH PRIVILEGES\"",
  onlyif  => "mysql -u' '",
  path    => "/usr/bin",
  require => Service["mysql"],
}
```

In this command, instead of using the `unless` parameter we use the opposite `onlyif` parameter, which will run the main command only if the exit code of the test command is zero. This test command tries to access MySQL

using an empty user. If it can connect, Puppet needs to execute the `DELETE ...` SQL command to remove the anonymous account. Finally, we will create a final `exec` resource to create the user with permission to access the online store's schema:

```
exec ...
package ...
file ...
service ...
exec ...
exec ...

exec { "store-user":
  unless  => "mysql -ustore -pstoresecret store_schema",
  command => "mysql -uroot -e \"GRANT ALL PRIVILEGES ON \
                                store_schema.* TO 'store'@'%' \
                                IDENTIFIED BY 'storesecret';\"",
  path    => "/usr/bin/",
  require => Exec["store-schema"],
}
```

When running Puppet for the last time in the database server, we will see an output like this:

```
$ vagrant provision db
==> db: Running provisioner: puppet...
Running Puppet with db.pp...
stdin: is not a tty
notice: /Stage[main]//Exec[apt-update]/returns: executed \
        successfully
notice: /Stage[main]//Exec[store-schema]/returns: executed \
        successfully
notice: /Stage[main]//Exec[store-user]/returns: executed \
        successfully
notice: /Stage[main]//Exec[remove-anonymous-user]/returns: \
        executed successfully
notice: Finished catalog run in 15.56 seconds
```

With that, we have fully restored the database server and all Nagios checks for the `db` host should go green. The only check that will remain red is the

Tomcat check, because the application is not yet available. In the next sections we will automate the process of provisioning the `web` server and the application deploy process. For now, the full contents of the `db.pp` manifest are:

```
exec { "apt-update":
  command => "/usr/bin/apt-get update"
}

package { "mysql-server":
  ensure  => installed,
  require => Exec["apt-update"],
}

file { "/etc/mysql/conf.d/allow_external.cnf":
  owner   => mysql,
  group   => mysql,
  mode    => 0644,
  content => template("/vagrant/manifests/allow_ext.cnf"),
  require => Package["mysql-server"],
  notify  => Service["mysql"],
}

service { "mysql":
  ensure     => running,
  enable     => true,
  hasstatus  => true,
  hasrestart => true,
  require    => Package["mysql-server"],
}

exec { "store-schema":
  unless  => "mysql -uroot store_schema",
  command => "mysqladmin -uroot create store_schema",
  path    => "/usr/bin/",
  require => Service["mysql"],
}

exec { "remove-anonymous-user":
```

```
  command => "mysql -uroot -e \"DELETE FROM mysql.user \
                                WHERE user=''; \
                                FLUSH PRIVILEGES\"",
  onlyif  => "mysql -u' '",
  path    => "/usr/bin",
  require => Service["mysql"],
}

exec { "store-user":
  unless  => "mysql -ustore -pstoresecret store_schema",
  command => "mysql -uroot -e \"GRANT ALL PRIVILEGES ON \
                                store_schema.* TO 'store'@'%' \
                                IDENTIFIED BY 'storesecret';\"",
  path    => "/usr/bin/",
  require => Exec["store-schema"],
}
```

4.4 Provisioning the web server

Now that the `db` server has been restored, we will automate the provisioning and deploy process in the `web` server. For that, we will create a new file in the `manifests` directory called `web.pp`:

```
.
 Vagrantfile
 manifests
     allow_ext.cnf
     db.pp
     web.pp
```

Instead of associating the new manifest to the existing `web` server, we will create a new temporary VM called `web2` to test our Puppet code in isolation before applying it to the production environment. At the end of the chapter, we will get rid of this `web2` VM and we will apply the same configuration to the actual `web` server. The contents of the `Vagrantfile` with the new VM must be:

```
VAGRANTFILE_API_VERSION = "2"
```

```
Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  config.vm.box = "hashicorp/precise32"

  config.vm.define :db do |db_config|
    db_config.vm.hostname = "db"
    db_config.vm.network :private_network,
                         :ip => "192.168.33.10"
    db_config.vm.provision "puppet" do |puppet|
      puppet.manifest_file = "db.pp"
    end
  end

  config.vm.define :web do |web_config|
    web_config.vm.hostname = "web"
    web_config.vm.network :private_network,
                          :ip => "192.168.33.12"
  end

  config.vm.define :web2 do |web_config|
    web_config.vm.hostname = "web2"
    web_config.vm.network :private_network,
                          :ip => "192.168.33.13"
    web_config.vm.provision "puppet" do |puppet|
      puppet.manifest_file = "web.pp"
    end
  end

  config.vm.define :monitor do |monitor_config|
    monitor_config.vm.hostname = "monitor"
    monitor_config.vm.network :private_network,
                              :ip => "192.168.33.14"
  end
end
```

Initially, we are just going to declare the resources to run the `apt-get update` command and install the base packages for the `web2` server, similar to the commands we executed in chapter 2:

```
exec { "apt-update":
  command => "/usr/bin/apt-get update"
}

package { ["mysql-client", "tomcat7"]:
  ensure  => installed,
  require => Exec["apt-update"],
}
```

When we bring up the new VM, Puppet will apply this configuration and install the packages:

```
$ vagrant up web2
Bringing machine 'web2' up with 'virtualbox' provider...
==> web2: Importing base box 'hashicorp/precise32'...
...
==> web2: Running provisioner: puppet...
Running Puppet with web.pp...
stdin: is not a tty
notice: /Stage[main]//Exec[apt-update]/returns: executed \
        successfully
notice: /Stage[main]//Package[mysql-client]/ensure: ensure \
        changed 'purged' to 'present'
notice: /Stage[main]//Package[tomcat7]/ensure: ensure changed \
        'purged' to 'present'
notice: Finished catalog run in 105.89 seconds
```

After doing that, you can open your browser and visit the URL http://192.168.33.13:8080/, where you should see a page saying "It Works!" That means that Tomcat is installed and running properly.

The next steps performed in chapter 2 were to configure a secure connection with HTTPS, to create the `keystore` to store the SSL certificate, and to the increase the amount of memory available to the Java virtual machine when Tomcat starts.

All these operations involve editing or creating a new file on the system. We already know how to do that with Puppet, so we will do all three steps at once. First, we'll use the configuration files that are already present in the `web` server as templates for our new Puppet manifest. The following commands will copy the files from the `web` VM to the Vagrant shared directory:

```
$ vagrant ssh web -- 'sudo cp /var/lib/tomcat7/conf/.keystore \
> /vagrant/manifests/'
$ vagrant ssh web -- 'sudo cp /var/lib/tomcat7/conf/server.xml \
> /vagrant/manifests/'
$ vagrant ssh web -- 'sudo cp /etc/default/tomcat7 \
> /vagrant/manifests/'
```

Instead of using the `vagrant ssh` command to login to the virtual machine, we pass a quoted command after the two dashes `--`. This command will run inside the VM. The `/vagrant` path is where Vagrant maps the working directory where the `Vagrantfile` is defined inside the VM. If everything goes well, the files on your host machine should look like:

```
.
 Vagrantfile
 manifests
     .keystore
     allow_ext.cnf
     db.pp
     server.xml
     tomcat7
     web.pp
```

Since we do not need to change anything in the content of these files, it is enough to declare new `file` resources so that Puppet will deliver them to the right place, adjusting owners and permissions:

```
exec ...
package ...

file { "/var/lib/tomcat7/conf/.keystore":
  owner   => root,
  group   => tomcat7,
  mode    => 0640,
  source  => "/vagrant/manifests/.keystore",
  require => Package["tomcat7"],
}

file { "/var/lib/tomcat7/conf/server.xml":
```

```
  owner   => root,
  group   => tomcat7,
  mode    => 0644,
  source  => "/vagrant/manifests/server.xml",
  require => Package["tomcat7"],
}

file { "/etc/default/tomcat7":
  owner   => root,
  group   => root,
  mode    => 0644,
  source  => "/vagrant/manifests/tomcat7",
  require => Package["tomcat7"],
}
```

Finally, we need to declare the `tomcat7` service that needs to be restarted when the configuration files are changed. The parameters of the `service` resource are the same as the ones we used in the `mysql` service in the previous section:

```
exec ...
package ...

file { "/var/lib/tomcat7/conf/.keystore":
  ...
  notify  => Service["tomcat7"],
}

file { "/var/lib/tomcat7/conf/server.xml":
  ...
  notify  => Service["tomcat7"],
}

file { "/etc/default/tomcat7":
  ...
  notify  => Service["tomcat7"],
}

service { "tomcat7":
```

```
  ensure     => running,
  enable     => true,
  hasstatus  => true,
  hasrestart => true,
  require    => Package["tomcat7"],
}
```

We can then run the entire manifest to apply the latest configuration changes to the `web2` server:

```
$ vagrant provision web2
==> web2: Running provisioner: puppet...
Running Puppet with web.pp...
stdin: is not a tty
notice: /Stage[main]//Exec[apt-update]/returns: executed \
        successfully
notice: /Stage[main]//File[ /var/lib/tomcat7/conf \
                            /server.xml]/ content: content \
        changed '{md5}523967040584a921450af2265902568d' to \
                '{md5}e4f0d5610575a720ad19fae4875e9f2f'
notice: /Stage[main]//File[ /var/lib/tomcat7/conf \
                            /.keystore]/ ensure: defined \
        content as '{md5}80792b4dc1164d67e1eb859c170b73d6'
notice: /Stage[main]//File[ /etc/default \
                            /tomcat7]/ content: content \
        changed '{md5}49f3fe5de425aca649e2b69f4495abd2' to \
        '{md5}1f429f1c7bdccd3461aa86330b133f51'
notice: /Stage[main]//Service[tomcat7]: Triggered 'refresh' \
        from 3 events
notice: Finished catalog run in 14.58 seconds
```

With that completed, we have Tomcat set up to accept both HTTP and HTTPS connections. We can test it by opening a new browser window and visiting the URL https://192.168.33.13:8443/. After accepting the self-signed certificate, you will see a page that says "It works!"

[Deploying the application]

To have a complete `web` server, we need to deploy the online store application. Instead of using Puppet to execute the build process inside the VM

– like we did in chapter 2 – we will reuse the `.war` artifact that has already been created. In chapter 6 we will revisit this decision and find a better way to create the application's `.war` artifact.

We will also reuse the `context.xml` file as a template, in which we define the JNDI resources to connect to the database. Using the same trick from the previous section, we can copy the files from the already running `web` VM:

```
$ vagrant ssh web -- 'sudo cp \
> /var/lib/tomcat7/conf/context.xml /vagrant/manifests/'
$ vagrant ssh web -- 'sudo cp \
> /var/lib/tomcat7/webapps/devopsnapratica.war \
> /vagrant/manifests/'
```

Now the directory structure in your host machine should be:

```
.
 Vagrantfile
 manifests
     .keystore
     allow_ext.cnf
     context.xml
     db.pp
     devopsnapratica.war
     server.xml
     tomcat7
     web.pp
```

The artifact `devopsnapratica.war` containing the web application doesn't need to be changed, but we need to define variables in the manifest with the information it will need to access the database. Those variables will be replaced when Puppet processes the ERB template while creating the `context.xml` file:

```
exec ...
package ...
file ...
file ...
file ...
service ...
```

```
$db_host     = "192.168.33.10"
$db_schema   = "store_schema"
$db_user     = "store"
$db_password = "storesecret"

file { "/var/lib/tomcat7/conf/context.xml":
  owner   => root,
  group   => tomcat7,
  mode    => 0644,
  content => template("/vagrant/manifests/context.xml"),
  require => Package["tomcat7"],
  notify  => Service["tomcat7"],
}

file { "/var/lib/tomcat7/webapps/devopsnapratica.war":
  owner   => tomcat7,
  group   => tomcat7,
  mode    => 0644,
  source  => "/vagrant/manifests/devopsnapratica.war",
  require => Package["tomcat7"],
  notify  => Service["tomcat7"],
}
```

The Puppet syntax for declaring variables is a dollar sign `$` followed by the name of the variable. We use the equal symbol `=` to assign values to our variables. In order to use these variables inside the `context.xml` template, we need to change its contents, replacing existing values with the tag `<%= var %>`, where `var` is the name of the variable without the dollar sign `$`. The new ERB template for the `manifests/context.xml` file will be:

```
<?xml version='1.0' encoding='utf-8'?>
<Context>
    <!-- Default set of monitored resources -->
    <WatchedResource>WEB-INF/web.xml</WatchedResource>

    <Resource name="jdbc/web" auth="Container"
      type="javax.sql.DataSource" maxActive="100"
      maxIdle="30"
```

```
      maxWait="10000"
      username="<%= db_user %>"
      password="<%= db_password %>"
      driverClassName="com.mysql.jdbc.Driver"
      url="jdbc:mysql://<%= db_host %>:3306/<%= db_schema %>"/>

    <Resource name="jdbc/secure" auth="Container"
      type="javax.sql.DataSource" maxActive="100" maxIdle="30"
      maxWait="10000"
      username="<%= db_user %>"
      password="<%= db_password %>"
      driverClassName="com.mysql.jdbc.Driver"
      url="jdbc:mysql://<%= db_host %>:3306/<%= db_schema %>"/>

    <Resource name="jdbc/storage" auth="Container"
      type="javax.sql.DataSource" maxActive="100" maxIdle="30"
      maxWait="10000"
      username="<%= db_user %>"
      password="<%= db_password %>"
      driverClassName="com.mysql.jdbc.Driver"
      url="jdbc:mysql://<%= db_host %>:3306/<%= db_schema %>"/>
</Context>
```

With that, we complete the Puppet manifest to provision and deploy the web application on the `web2` server. We can do that by running the command:

```
$ vagrant provision web2
==> web2: Running provisioner: puppet...
Running Puppet with web.pp...
stdin: is not a tty
notice: /Stage[main]//Exec[apt-update]/returns: executed \
        successfully
notice: /Stage[main]//File[ /var/lib/tomcat7/conf \
                            /context.xml]/ content: content \
        changed '{md5}4861cda2bbf3a56fbfdb78622c550019' to \
                '{md5}8365f14d15ddf99b1d20f9672f0b98c7'
notice: /Stage[main]//File[ /var/lib/tomcat7/webapps \
                            /devopsnapratica.war]/ ensure: \
```

```
        defined content as \
        '{md5}cba179ec04e75ce87140274b0aaa2263'
notice: /Stage[main]//Service[tomcat7]: Triggered 'refresh' \
        from 2 events
notice: Finished catalog run in 21.78 seconds
```

Now try to access the URL http://192.168.33.13:8080/devopsnapratica/ in your browser to see if the online store is working correctly. Success!

Now that we know how to restore the `web` server completely, we can get rid of the `web2` VM. To demonstrate the power of having an automated process of provisioning and deploy, we will also destroy the `web` VM and recreate it from scratch:

```
$ vagrant destroy web web2
    web2: Are you sure you want to destroy the 'web2' VM?[y/N] y
==> web2: Forcing shutdown of VM...
==> web2: Destroying VM and associated drives...
==> web2: Running cleanup tasks for 'puppet' provisioner...
    web: Are you sure you want to destroy the 'web' VM? [y/N] y
==> web: Forcing shutdown of VM...
==> web: Destroying VM and associated drives...
```

Before restarting the virtual machine, we need to update the Vagrant configuration file – the `Vagrantfile` – to remove the `web2` server and leave only the section for the `web` server:

```
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  config.vm.box = "hashicorp/precise32"

  config.vm.define :db do |db_config|
    db_config.vm.hostname = "db"
    db_config.vm.network :private_network,
                         :ip => "192.168.33.10"
    db_config.vm.provision "puppet" do |puppet|
      puppet.manifest_file = "db.pp"
    end
  end
```

```
  config.vm.define :web do |web_config|
    web_config.vm.hostname = "web"
    web_config.vm.network :private_network,
                          :ip => "192.168.33.12"
    web_config.vm.provision "puppet" do |puppet|
      puppet.manifest_file = "web.pp"
    end
  end

  config.vm.define :monitor do |monitor_config|
    monitor_config.vm.hostname = "monitor"
    monitor_config.vm.network :private_network,
                              :ip => "192.168.33.14"
  end
end
```

The last step is to rebuild the `web` VM from scratch:

```
$ vagrant up web
Bringing machine 'web' up with 'virtualbox' provider...
==> web: Importing base box 'hashicorp/precise32'...
==> web: Matching MAC address for NAT networking...
==> web: Checking if box 'hashicorp/precise32' is up to date...
==> web: Setting the name of the VM:
         blank_web_1394856878524_4521
==> web: Fixed port collision for 22 => 2222. Now on port 2200.
==> web: Clearing any previously set network interfaces...
==> web: Preparing network interfaces based on configuration...
    web: Adapter 1: nat
    web: Adapter 2: hostonly
==> web: Forwarding ports...
    web: 22 => 2200 (adapter 1)
==> web: Running 'pre-boot' VM customizations...
==> web: Booting VM...
==> web: Waiting for machine to boot. This may take a few
         minutes...
    web: SSH address: 127.0.0.1:2200
    web: SSH username: vagrant
    web: SSH auth method: private key
```

```
==> web: Machine booted and ready!
==> web: Configuring and enabling network interfaces...
==> web: Mounting shared folders...
    web: /vagrant => /private/tmp/blank
    web: /tmp/vagrant-puppet-2/manifests =>
         /private/tmp/blank/manifests
==> web: Running provisioner: puppet...
Running Puppet with web.pp...
stdin: is not a tty
notice: /Stage[main]//Exec[apt-update]/returns: executed \
        successfully
notice: /Stage[main]//Package[mysql-client]/ensure: ensure \
        changed 'purged' to 'present'
notice: /Stage[main]//Package[tomcat7]/ensure: ensure \
        changed 'purged' to 'present'
notice: /Stage[main]//File[ /var/lib/tomcat7/conf \
                          /context.xml]/content: content \
        changed '{md5}4861cda2bbf3a56fbfdb78622c550019' to \
        '{md5}8365f14d15ddf99b1d20f9672f0b98c7'
notice: /Stage[main]//File[ /var/lib/tomcat7/conf \
                          /server.xml]/content: content \
        changed '{md5}523967040584a921450af2265902568d' to \
        '{md5}e4f0d5610575a720ad19fae4875e9f2f'
notice: /Stage[main]//File[ /var/lib/tomcat7/conf \
                          /.keystore]/ensure: defined \
        content as '{md5}80792b4dc1164d67e1eb859c170b73d6'
notice: /Stage[main]//File[ /var/lib/tomcat7/webapps \
                          /devopsnapratica.war]/ensure: \
        defined content as \
        '{md5}cba179ec04e75ce87140274b0aaa2263'
notice: /Stage[main]//File[ /etc/default \
                          /tomcat7]/content: content \
        changed '{md5}49f3fe5de425aca649e2b69f4495abd2' to \
        '{md5}1f429f1c7bdccd3461aa86330b133f51'
notice: /Stage[main]//Service[tomcat7]: Triggered 'refresh' \
        from 5 events
notice: Finished catalog run in 106.31 seconds
```

Done! We managed to rebuild the `web` server and deploy the online store **in 1 minute!**

All Nagios checks should turn green again so that the online store is once more available to our users. The entire `web.pp` manifest, by the end of this chapter looks like:

```
exec { "apt-update":
  command => "/usr/bin/apt-get update"
}

package { ["mysql-client", "tomcat7"]:
  ensure  => installed,
  require => Exec["apt-update"],
}

file { "/var/lib/tomcat7/conf/.keystore":
  owner   => root,
  group   => tomcat7,
  mode    => 0640,
  source  => "/vagrant/manifests/.keystore",
  require => Package["tomcat7"],
  notify  => Service["tomcat7"],
}

file { "/var/lib/tomcat7/conf/server.xml":
  owner   => root,
  group   => tomcat7,
  mode    => 0644,
  source  => "/vagrant/manifests/server.xml",
  require => Package["tomcat7"],
  notify  => Service["tomcat7"],
}

file { "/etc/default/tomcat7":
  owner   => root,
  group   => root,
  mode    => 0644,
  source  => "/vagrant/manifests/tomcat7",
  require => Package["tomcat7"],
```

```
  notify  => Service["tomcat7"],
}

service { "tomcat7":
  ensure     => running,
  enable     => true,
  hasstatus  => true,
  hasrestart => true,
  require    => Package["tomcat7"],
}

$db_host     = "192.168.33.10"
$db_schema   = "store_schema"
$db_user     = "store"
$db_password = "storesecret"

file { "/var/lib/tomcat7/conf/context.xml":
  owner   => root,
  group   => tomcat7,
  mode    => 0644,
  content => template("/vagrant/manifests/context.xml"),
  require => Package["tomcat7"],
  notify  => Service["tomcat7"],
}

file { "/var/lib/tomcat7/webapps/devopsnapratica.war":
  owner   => tomcat7,
  group   => tomcat7,
  mode    => 0644,
  source  => "/vagrant/manifests/devopsnapratica.war",
  require => Package["tomcat7"],
  notify  => Service["tomcat7"],
}
```

Investing in automation allowed us to destroy and rebuild our servers in a matter of minutes. You must have also noticed that we did not spend much time executing manual commands in our servers. We have transferred the responsibility of executing those commands to Puppet and declared the desired state of our servers in manifest files.

The Puppet code is starting to grow and we don't have a clear structure for organizing our manifest files and templates. We are also treating the `war` artifact as a static file, something we know is not ideal. In the next chapters, we will revisit these decisions and create a more robust ecosystem for doing continuous delivery of the online store.

Chapter 5

Puppet beyond the basics

By the end of chapter 4, the Puppet code to configure the online store's infrastructure is not very well organized. The only separation that we did was creating two manifest files, one for each server: `web` and `db`. However, in each file, the code is simply a long list of resources. In addition, the configuration and template files are written without any structure. In this chapter we will learn new Puppet concepts and features while refactoring the online store infrastructure code to make it more idiomatic and better factored.

5.1 Classes and defined types

Puppet manages a single instance of each resource defined in a manifest, making them similar to a *singleton*. Likewise, a **class** is a collection of singleton resources in the system. If you know object-oriented languages, do not get confused with this terminology. A Puppet class cannot be instantiated multi-

ple times. Classes are just a way of giving a name to a collection of resources that will be applied as a unit.

A good use case for Puppet classes is when you are configuring services that you need to install in the system only once. For example, in the `db.pp` file we install and setup MySQL server and then create the application-specific user and schema for the online store. In a real scenario, we could have several schemas and users in the same underlying database, but we do not install MySQL multiple times in the same system. MySQL server is a good candidate to be setup in a generic class, which we will call `mysql-server`. Refactoring our `db.pp` file to declare and use this new class, we will have:

```
class mysql-server {
  exec    { "apt-update": ... }
  package { "mysql-server": ... }
  file    { "/etc/mysql/conf.d/allow_external.cnf": ... }
  service { "mysql": ... }
  exec    { "remove-anonymous-user": ... }
}

include mysql-server

exec { "store-schema": ...,
  require => Class["mysql-server"],
}
exec { "store-user": ... }
```

Notice that we moved the `Exec["remove-anonymous-user"]` resource – that revokes access to the anonymous user – into the `mysql-server` class because this is something that should happen only once when MySQL server is installed. Another change you may notice is that the `Exec["store-schema"]` resource now depends on `Class["mysql-server"]` instead of the previous `Service["mysql"]` resource. This is a way to encapsulate implementation details within a class, isolating this knowledge from the rest of the code, which can declare dependencies on something more abstract and stable.

To define a new class, just choose its name and define all of its resources in a `class <class name> { ... }` declaration. To use a class, you can

use the `include` syntax or a version that's more similar to how we have been defining other resources: `class { "<class name>": ... }`.

On the other hand, the resources to create the online store schema and user can be reused to create schemas and users for other applications running in the same database server. Putting them in a class would be the wrong way to encapsulate them, because Puppet would force them to be singletons. For situations like this, Puppet has another form of encapsulation known as "defined types."

A **defined type** is a collection of resources that can be used multiple times in the same manifest. They help you eliminate duplication by grouping related resources that can be reused together. You can think of them as the equivalent to macros in a programming language. Moreover, they can be parameterized and define default values for optional parameters. Grouping the two `exec` resources that create the database schema and user in a defined type called `mysql-db`, we will have:

```
class mysql-server { ... }

define mysql-db($schema, $user = $title, $password) {
  Class['mysql-server'] -> Mysql-db[$title]

  exec { "$title-schema":
    unless  => "mysql -uroot $schema",
    command => "mysqladmin -uroot create $schema",
    path    => "/usr/bin/",
  }

  exec { "$title-user":
    unless  => "mysql -u$user -p$password $schema",
    command => "mysql -uroot -e \"GRANT ALL PRIVILEGES ON \
                                  $schema.* TO '$user'@'%' \
                                  IDENTIFIED BY '$password';\"",
    path    => "/usr/bin/",
    require => Exec["$title-schema"],
  }
}
```

```
include mysql-server

mysql-db { "store":
  schema   => "store_schema",
  password => "storesecret",
}
```

The syntax to declare a defined type is `define <defined type name>(<parameters>) { ... }`. In our example, the `mysql-db` defined type accepts three parameters: `$schema`, `$user` and `$password`. The `$user` parameter, unless otherwise specified, will take the default value of the special `$title` parameter. In order to understand the value of `$title` parameter – which does not need to be explicitly declared – just look at the syntax used when instantiating a defined type: `mysql-db { "store": schema => "store_schema", ... }`. The resource name that instantiates the defined type, in this case `store`, will be passed as the value for the `$title` parameter. The other parameters are passed following the same syntax used in other native Puppet resources: the parameter name and its value separated by an arrow `=>`.

We moved the two `exec` resources that create the schema and the user to a defined type. We also replaced all the hard-coded references with the respective parameters, paying attention to use double quotes in strings so Puppet can expand the values correctly. In order to use the defined type more than once, we need to parameterize the names of the `exec` resources to make them unique, using the `$title` parameter. The last update was to promote the dependency with the `mysql-server` class to the top of the defined type. With that, the individual resources inside the defined type don't have to declare any dependencies with external resources, making our code easier to maintain in the future.

Using classes and defined types, we managed to refactor our Puppet code to make it more reusable. However, we have not yet changed how the files are organized. Everything continues to be declared in a single file. We need to learn a better way to organize our files.

5.2 Using modules for packaging and distribution

Puppet has a standard for packaging and structuring your code: They are called **modules.** Modules define a standard directory structure in which you should place your files, as well as some naming conventions. Modules are also a way to share Puppet code with the community. The Puppet Forge (http://forge.puppetlabs.com/) is a website maintained by Puppet Labs where you can find many modules written by the community, or you can register and share a module you wrote.

Depending on your experience with different languages, the equivalent of a Puppet module in Ruby, Java, Python and .NET would be a *gem*, a *jar*, an *egg* and a *DLL*, respectively. The simplified directory structure for a Puppet module is:

```
<module name>/
 files
    ...
 manifests
    init.pp
 templates
    ...
 tests
     init.pp
```

First of all, the name of the root directory defines the module name. The most important directory is the *manifests* directory, because it is where you place your manifest files (with a `.pp` extension). Inside it, there must be at least one file called `init.pp`, which is loaded as the entry point for the module. The *files* directory contains static configuration files that can be used by a `file` resource in a manifest using a special URL: `puppet:///modules/<module name>/<file>`. The *templates* directory contains ERB files that can be referenced in a manifest using the module name: `template('<module name>/<ERB file>')`.

Finally, the *tests* directory contains examples of how to use the Classes, as well as defined types exposed by the module. These tests do not per-

form any kind of automated verification. You can run them using the command `puppet apply --noop`. This command simulates a Puppet execution without performing any changes in the system. If any syntax or compilation errors are found, you will detect them in output of the `puppet apply --noop` command.

With that knowledge, we will perform a preparatory refactoring prior to creating our own module. Let's first split the `mysql-server` class and the `mysql-db` defined type definitions into two separate files called `server.pp` and `db.pp`, respectively. In the same level where we currently have the `Vagrantfile`, we will create a new `modules` directory where we will create our modules. Inside it, we will make a new module for installing and configuring MySQL, called `mysql`, creating the following directory structure and moving the respective files there:

```
.
 Vagrantfile
 manifests
    .keystore
    context.xml
    db.pp
    devopsnapratica.war
    server.xml
    tomcat7
    web.pp
 modules
     mysql
         manifests
            init.pp
            db.pp
            server.pp
         templates
            allow_ext.cnf
```

Note that we also created a new file called `init.pp` in the `modules/mysql/manifests` directory. This file will be the entry point for our module and it must declare an empty `mysql` class that serves as a namespace for the rest of the module:

```
class mysql { }
```

We also move the `allow_ext.cnf` file to the `modules/mysql/templates` directory. Since we are following the standard module structure, we no longer need to reference the template using an absolute path. Puppet will accept a path of the form `<module name>/<file>` and knows how to find the `<file>` template inside the selected module. We can then change the `file` resource declared on the `modules/mysql/manifests/server.pp` manifest from `template("/vagrant/manifests/allow_ext.cnf")` to `template("mysql/allow_ext.cnf")`:

```
class mysql::server {
  exec { "apt-update": ... }
  package { "mysql-server": ... }

  file { "/etc/mysql/conf.d/allow_external.cnf":
    ...
    content => template("mysql/allow_ext.cnf"),
    ...
  }

  service { "mysql": ... }
  exec { "remove-anonymous-user": ... }
}
```

Note that we renamed the `mysql-server` class to `mysql::server`. This is the convention that Puppet uses to namespace classes and types inside of a module. Likewise, we should also rename the `mysql-db` defined type to `mysql::db` in the `modules/mysql/manifests/db.pp` file, and update its dependency with the `mysql::server` class:

```
define mysql::db($schema, $user = $title, $password) {
  Class['mysql::server'] -> Mysql::Db[$title]
  ...
}
```

Finally, we can greatly simplify the `manifests/db.pp` file to use the

`mysql::server` class and the `mysql::db` defined type from our new module:

```
include mysql::server

mysql::db { "store":
  schema   => "store_schema",
  password => "storesecret",
}
```

With this, we create a generic module that provides a class to install the MySQL server and a defined type to create an application-specific schema and user. It may seem confusing that we are moving things around without changing any behavior, however, that is exactly the purpose of refactoring code! We improve the code's structure without changing its external behavior. This is beneficial because it makes it more maintainable, isolating individual components and group concepts that are related. Principles of software design – such as encapsulation, coupling and cohesion – also help to improve infrastructure code.

In order to test that everything still works, you can destroy and recreate the `db` VM again, but before that we have to make a small change to tell Vagrant that we have a new module that needs to be loaded when Puppet runs. We change the `Vagrantfile` file to add the `module_path` configuration option in the `web` and `db` server definition:

```
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  ...
  config.vm.define :db do |db_config|
    ...
    db_config.vm.provision "puppet" do |puppet|
      puppet.module_path = "modules"
      puppet.manifest_file = "db.pp"
    end
  end

  config.vm.define :web do |web_config|
```

```
    ...
    web_config.vm.provision "puppet" do |puppet|
      puppet.module_path = "modules"
      puppet.manifest_file = "web.pp"
    end
  end

  ...
end
```

Now that the Puppet code for managing the database infrastructure is refactored, it is time to improve the code that installs, configures and deploys our application on the `web` server.

5.3 Refactoring the web server Puppet code

Now that we know how to refactor our Puppet code, let's create a new module called `tomcat` that will be used by the `web` server.

First, we create the directory structure of the new module and move the files from the `manifests` directory into the following module directories inside `modules/tomcat`:

```
.
 Vagrantfile
 manifests
    .keystore
    db.pp
    devopsnapratica.war
    web.pp
 modules
     mysql
        ...
     tomcat
         files
            server.xml
            tomcat7
         manifests
         templates
             context.xml
```

The only files that will remain in the `manifests` directory are the server manifests (`db.pp` and `web.pp`), the WAR file containing the online store application, and the `.keystore` file containing its SSL certificate. The reason for doing this is to follow an important software development principle: **separation of concerns**. If we look at what's defined in the `web.pp` file, we are mixing generic resources – that install and configure Tomcat – with specific resources for the online store. If we simply move everything to the same module, it will not be reusable. Think about it from an external perspective, such as another team in your company or the community at large: A module that installs and configures Tomcat is much more useful than a module that configures Tomcat and the entire online store application.

Keeping this in mind, the goal of our refactoring is still to extract a Tomcat module that does not reference anything specific in the online store. As we move resource definitions from the `web.pp` manifest to the `tomcat` module, we'll keep things that are specific to the online store on the server's manifest and move the more generic resources to the `tomcat` module.

Before we start tackling the Tomcat features, let's start with something slightly simpler: extracting the part that installs the MySQL client to a new class in the `mysql` module. We create a new `client.pp` file under `modules/mysql/manifests`, with the contents:

```
class mysql::client {
  exec { "apt-update":
    command => "/usr/bin/apt-get update"
  }

  package { "mysql-client":
    ensure  => installed,
    require => Exec["apt-update"],
  }
}
```

The motivation for doing this is, again, separation of concerns. The `mysql::client` class can be useful in other contexts, even when you are not developing a web application, therefore putting it in the `mysql` module makes more sense than in the `tomcat` module. We are gradually starting to untangle our code.

With that, we can start editing the `web.pp` file. We will include the new `mysql::client` class, remove the `Exec[apt-update]` resource – which has migrated to the new class – and update all the references to static and template files using the new path within the module:

```
include mysql::client

package { "tomcat7": ... }

file { "/var/lib/tomcat7/conf/.keystore": ... }

file { "/var/lib/tomcat7/conf/server.xml": ...
  source  => "puppet:///modules/tomcat/server.xml", ...
}

file { "/etc/default/tomcat7": ...
  source  => "puppet:///modules/tomcat/tomcat7", ...
}

service { "tomcat7": ... }

$db_host     = "192.168.33.10"
$db_schema   = "store_schema"
$db_user     = "store"
$db_password = "storesecret"

file { "/var/lib/tomcat7/conf/context.xml": ...
  content => template("tomcat/context.xml"), ...
}

file { "/var/lib/tomcat7/webapps/devopsnapratica.war": ... }
```

The only references that should continue using the absolute paths are those in the `File[/var/lib/tomcat7/conf/.keystore]` and `File[/var/lib/tomcat7/webapps/devopsnapratica.war]` resources, since we have not moved them inside the module.

Now, we can start to move parts of the code inside the `web.pp` file to new classes and defined types within the `tomcat` module. We will create a

new `server.pp` file inside the `modules/tomcat/manifests` directory and move the following resources from the `web.pp` file to a new class that we will call `tomcat::server`:

```
class tomcat::server {
  package { "tomcat7": ... }
  file { "/etc/default/tomcat7": ... }
  service { "tomcat7": ... }
}
```

To expose this new class in the new module, we also need to create an `init.pp` file inside the same directory, which will initially contain a single empty class: `class tomcat { }`. With that, we can change the `web.pp` file, removing resources that were moved and replacing them with an `include` of the newly created `tomcat::server` class:

```
include mysql::client
include tomcat::server

file { "/var/lib/tomcat7/conf/.keystore": ... }
file { "/var/lib/tomcat7/conf/server.xml": ... }

$db_host     = "192.168.33.10"
$db_schema   = "store_schema"
$db_user     = "store"
$db_password = "storesecret"

file { "/var/lib/tomcat7/conf/context.xml": ... }
file { "/var/lib/tomcat7/webapps/devopsnapratica.war": ... }
```

Notice that we have not yet moved all resources of the type `file` to the new class, because they still have references and settings that are specific to our application:

- The `.keystore` file stores the SSL certificate of the online store. It's not a good idea to distribute our certificate in a generic Tomcat module that other people or teams can reuse.

- The `server.xml` file has settings to enable HTTPS containing sensitive information that is also specific to the online store.
- The `context.xml` file configures the JNDI resources for accessing the online store's database schema.
- Finally, the `devopsnapratica.war` file contains the entire online store application.

In order to make the Tomcat module more generic, let's first solve the problem with the SSL settings and the `.keystore` file. In the `server.xml` configuration file, instead of having the path to the keystore and passwords hard-coded, let's turn it into a template and pass this information as parameters to the `tomcat::server` class. To understand where we are trying to go, we will look at the changes to the `web.pp` file:

```
include mysql::client

$keystore_file = "/etc/ssl/.keystore"
$ssl_connector = {
  "port"         => 8443,
  "protocol"     => "HTTP/1.1",
  "SSLEnabled"   => true,
  "maxThreads"   => 150,
  "scheme"       => "https",
  "secure"       => "true",
  "keystoreFile" => $keystore_file,
  "keystorePass" => "secret",
  "clientAuth"   => false,
  "sslProtocol"  => "SSLv3",
}
$db_host     = "192.168.33.10"
$db_schema   = "store_schema"
$db_user     = "store"
$db_password = "storesecret"

file { $keystore_file:
  mode    => 0644,
  source  => "/vagrant/manifests/.keystore",
```

```
}

class { "tomcat::server":
  connectors => [$ssl_connector],
  require    => File[$keystore_file],
}

file { "/var/lib/tomcat7/conf/context.xml": ... }
file { "/var/lib/tomcat7/webapps/devopsnapratica.war": ... }
```

We will discuss each of the changes: first, we changed the `.keystore` file directory to a more generic place, outside of Tomcat, and store its location in a new `$keystore_file` variable. With that, we removed all the references and dependencies with Tomcat from the `File[/etc/ssl/.keystore]` resource. The second important change is that we have parameterized the `tomcat::server` class, passing a new parameter called `connectors`, which represents an array of connectors that need to be configured.

This example also serves to show two new data structures recognized by Puppet: arrays and hashes. An array is an indexed list of items, while a hash is a kind of dictionary that associates a key to a value. By passing the connectors list as a parameter, we can move the `File[/var/lib/tomcat7/conf/server.xml]` resource to the `tomcat::server` class, in the `modules/tomcat/manifests/server.pp` file:

```
class tomcat::server($connectors = []) {
  package { "tomcat7": ... }
  file { "/etc/default/tomcat7": ... }

  file { "/var/lib/tomcat7/conf/server.xml":
    owner   => root,
    group   => tomcat7,
    mode    => 0644,
    content => template("tomcat/server.xml"),
    require => Package["tomcat7"],
    notify  => Service["tomcat7"],
  }
```

```
  service { "tomcat7": ... }
}
```

Note that the class declaration syntax has changed to accept the new `$connectors` parameter, which has a default value corresponding to an empty list. Since the content of `server.xml` file needs to be dynamic, we need to move it from the `modules/tomcat/files` directory to the `modules/tomcat/templates` directory, also changing the file resource definition to use a template. Finally, we need to change the contents of the `server.xml` file to dynamically configure the connectors:

```
<?xml version='1.0' encoding='utf-8'?>
<Server port="8005" shutdown="SHUTDOWN">
  ...
  <Service name="Catalina">
    ...
    <Connector port="8080" protocol="HTTP/1.1"
               connectionTimeout="20000"
               URIEncoding="UTF-8"
               redirectPort="8443" />

    <%- connectors.each do |c| -%>
      <Connector
          <%= c.sort.map{|k,v| "#{k}='#{v}'"}.join(" ") %> />
    <%- end -%>
    ...
  </Service>
</Server>
```

Now we have removed any application-specific reference from the connector template. Using some Ruby in the `server.xml` template, we managed to turn what was once hardcoded into data that is passed in to the template. The interesting part is the line: `c.sort.map{|k,v| "#{k}='#{v}'"}.join(' ')`. The `sort` method guarantees that we will always iterate through the hash in the same order. The `map` operation transforms each key-value pair of the hash into a string formatted as an XML attribute. For example: a hash such as `{"port" => "80", "scheme" => "http"}` will be converted to the following list `["port='80'",`

`"scheme='http'"]`. The `join` operation concatenates all these strings into a single string, separating them with a space. The end result is the string: `"port='80' scheme='http"'`.

Now that we solved the problem of SSL configuration, we can use the same technique to move the `File[/var/lib/tomcat7/conf/context.xml]` resource to the `tomcat::server` class. Again we start with the desired change in the `web.pp` file:

```
include mysql::client

$keystore_file = "/etc/ssl/.keystore"
$ssl_conector  = ...
$db = {
  "user"     => "store",
  "password" => "storesecret",
  "driver"   => "com.mysql.jdbc.Driver",
  "url"      => "jdbc:mysql://192.168.33.10:3306/store_schema",
}

file { $keystore_file: ... }

class { "tomcat::server":
  connectors   => [$ssl_conector],
  data_sources => {
    "jdbc/web"     => $db,
    "jdbc/secure"  => $db,
    "jdbc/storage" => $db,
  },
  require      => File[$keystore_file],
}

file { "/var/lib/tomcat7/webapps/devopsnapratica.war": ... }
```

Following the same pattern, we added a new `data_sources` parameter to the `tomcat::server` class and passed a hash containing the configuration for each data source. We also moved the `File[/var/lib/tomcat7/conf/context.xml]` resource to the `tomcat::server` class in the

`modules/tomcat/manifests/server.pp` file:

```
class tomcat::server($connectors = [], $data_sources = []) {
  package { "tomcat7": ... }
  file { "/etc/default/tomcat7": ... }
  file { "/var/lib/tomcat7/conf/server.xml": ... }

  file { "/var/lib/tomcat7/conf/context.xml":
    owner   => root,
    group   => tomcat7,
    mode    => 0644,
    content => template("tomcat/context.xml"),
    require => Package["tomcat7"],
    notify  => Service["tomcat7"],
  }

  service { "tomcat7": ... }
}
```

We also need to change the `context.xml` template file to generate the JNDI resources dynamically:

```
<?xml version='1.0' encoding='utf-8'?>
<Context>
    <!-- Default set of monitored resources -->
    <WatchedResource>WEB-INF/web.xml</WatchedResource>

    <%- data_sources.sort.each do |data_source, db| -%>
      <Resource name="<%= data_source %>" auth="Container"
            type="javax.sql.DataSource" maxActive="100"
            maxIdle="30"
            maxWait="10000"
            username="<%= db['user'] %>"
            password="<%= db['password'] %>"
            driverClassName="<%= db['driver'] %>"
            url="<%= db['url'] %>"/>
    <%- end -%>
</Context>
```

We have refactored our code a lot, making it more organized and easier to

reuse. Right now, the directory structure with the new modules should look like:

```
.
 Vagrantfile
 manifests
    .keystore
    db.pp
    devopsnapratica.war
    web.pp
 modules
     mysql
        manifests
           client.pp
           db.pp
           init.pp
           server.pp
        templates
            allow_ext.cnf
     tomcat
         files
            tomcat7
         manifests
            init.pp
            server.pp
         templates
             context.xml
             server.xml
```

5.4 : Separation of concerns: infrastructure vs. application

So far, we have been forcing ourselves to write generic and reusable modules. However, there is still a lot of code left in the `web.pp` and `db.pp` manifests, as well as the loose `.keytore` and `devopsnapratica.war` files, which still have a bad smell.. A common practice to solve this problem is to create an application-specific module. But doesn't that contradict the arguments we have been making up until now?

Separating infrastructure modules from application modules is a way to apply the separation of concerns principle at another level. This is a form of composition: the application modules use the infrastructure modules, which are still reusable and independent. Figure 5.1 shows the desired structure and dependencies between our modules.

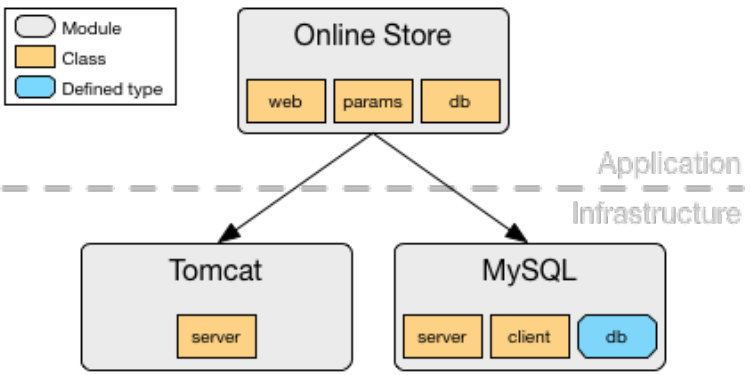

Fig. 5.1: Application modules vs. infrastructure modules

The last module we will create in this chapter will be responsible for installing and configuring the online store. For this, we need to expand our directory and file structure in the same way we did before, but this time for a new module called `online_store`:

```
.
 Vagrantfile
 manifests
    db.pp
    web.pp
 modules
     mysql
        ...
     online_store
        files
           .keystore
           devopsnapratica.war
        manifests
```

```
            db.pp
            init.pp
            params.pp
            web.pp
    tomcat
        ...
```

We moved the `.keystore` and `devopsnapratica.war` files to the `modules/online_store/files` directory and we created four manifest files to place the new classes of the `online_store` module: `db.pp`, `init.pp`, `params.pp` and `web.pp`. The `init.pp` file simply declares an empty class to serve as the root namespace for the rest of the module:

```
class online_store { }
```

The `db.pp` file will define a new `online_store::db` class, with its contents moved almost with no change from the `manifests/db.pp` file:

```
class online_store::db {
  include mysql::server
  include online_store::params

  mysql::db { $online_store::params::db['user']:
    schema   => $online_store::params::db['schema'],
    password => $online_store::params::db['password'],
  }
}
```

The only difference is that we moved the values for the database user, password, and schema variables to another `online_store::params` class in the same module. This is a common pattern in Puppet modules—centralizing parameters in a single class—so you don't have to duplicate them across the module.

The parameters defined in the `online_store::params` class also respect the new namespace. That is why we need to reference them using the full path `$online_store::params::db['user']` instead of just `$db['user']`. To better understand how these parameters have been defined, just look at the contents of the `params.pp` file, which defines the new `online_store::params` class:

```
class online_store::params {
  $keystore_file = "/etc/ssl/.keystore"

  $ssl_connector = {
    "port"          => 8443,
    "protocol"      => "HTTP/1.1",
    "SSLEnabled"    => true,
    "maxThreads"    => 150,
    "scheme"        => "https",
    "secure"        => "true",
    "keystoreFile" => $keystore_file,
    "keystorePass" => "secret",
    "clientAuth"    => false,
    "sslProtocol"   => "SSLv3",
  }

  $db = {
    "user"     => "store",
    "password" => "storesecret",
    "schema"   => "store_schema",
    "driver"   => "com.mysql.jdbc.Driver",
    "url"      => "jdbc:mysql://192.168.33.10:3306/",
  }
}
```

This is almost the same contents that we had in the `manifests/web.pp` file. The only difference is the addition of the `schema` key to the `$db` hash, which was previously being hardcoded at the end of the `$db['url']` parameter.

We did that to be able to reuse the same parameter structure in both the `online_store::db` and `online_store::web` classes. With this change, you also need to change the `modules/tomcat/templates/context.xml` template file to use a new `schema` key:

```
...
<Context>
  ...
```

```
  <Resource ... url="<%= db['url'] + db['schema'] %>"/>
  ...
</Context>
```

Since all parameters are now defined in a single class, it is easy to understand what code has moved and changed in the new `online_store::web` class:

```
class online_store::web {
  include mysql::client
  include online_store::params

  file { $online_store::params::keystore_file:
    mode    => 0644,
    source  => "puppet:///modules/online_store/.keystore",
  }

  class { "tomcat::server":
    connectors => [$online_store::params::ssl_connector],
    data_sources => {
      "jdbc/web"     => $online_store::params::db,
      "jdbc/secure"  => $online_store::params::db,
      "jdbc/storage" => $online_store::params::db,
    },
    require => File[$online_store::params::keystore_file],
  }

  file { "/var/lib/tomcat7/webapps/devopsnapratica.war":
    owner   => tomcat7,
    group   => tomcat7,
    mode    => 0644,
    source  => "puppet:///modules/online_store/
                devopsnapratica.war",
    require => Package["tomcat7"],
    notify  => Service["tomcat7"],
  }
}
```

Like we did with the `online_store::db` class, we also include the `online_store::params` class and use the full path for all parameters. We

also changed the `file` resources to reference the files using a relative path to the module instead of the full absolute path.

With that, our new module is complete and what is left in the manifest files for each server is now greatly simplified:

```
# Contents of file manifests/web.pp:
include online_store::web

# Contents of file manifests/db.pp:
include online_store::db
```

5.5 Puppet forge: reusing community modules

As the application development progresses and the system grows, it is also common to increase the number of classes, packages, and libraries used in the project. With that, a new concern arises: **dependency management**. This is such a common problem that many development communities created specific tools to solve it. In the Ruby community, Bundler manages dependencies between *gems*; in the Java community, Maven and Ivy manage *jars* and their dependencies; even system administrators use packages managers like `dpkg` or `rpm` to take care of dependencies when installing software.

The online store infrastructure code is also starting to show dependency management problems. We already have three Puppet modules and created an unwanted dependency between resources in the `mysql` and `tomcat` modules.

If you pay attention, the `Exec[apt-update]` resource is declared twice in the `mysql::server` and `mysql::client` classes, and it is used in three different places: in their own classes, and in the `tomcat::server` class.

In general, dependencies are easier to manage when the declaration and usage are closer to one other. A dependency between resources declared in the same file is better than a dependency between resources declared in different files; a dependency between different files on the same module is better than a dependency between resources from different modules.

In the current state of the code, the `tomcat` module has a direct dependency on a specific resource of the `mysql` module. If the online store decides

to stop using the `mysql` module, it will not be possible to install and configure Tomcat. This kind of dependency creates a strong coupling between the `mysql` and `tomcat` modules, making it difficult to use them independently.

Ideally, we would not have to declare any resource specific to APT. The only reason we declared this resource in chapter 4 was to update the APT index – running an `apt-get update` command – before installing any package. This is not a requirement of either MySQL or Tomcat, but a problem that appears when Puppet tries to install the first package in the system.

To improve our code and remove this unwanted dependency, we will spend some time learning how to use community modules. Let's use a PuppetLabs module that was written to handle APT in a more generic way.

Reading the module documentation (https://github.com/puppetlabs/puppetlabs-apt) , we find that it exposes the `apt` class which has a `always_apt_update` parameter that does exactly what we need:

```
class { 'apt':
  always_apt_update => true,
}
```

With this, we ensure that Puppet will run the `apt-get update` command every time it runs. However, we must ensure that the command runs before trying to install any package in the system. To declare this dependency, we are going to learn a new Puppet syntax:

```
Class['apt'] -> Package <| |>
```

The arrow `->` imposes a restriction on the execution order of resources. In the same way that the `require` parameter indicates a dependency between two resources, the arrow `->` ensures the resource on the left side – in this case the `apt` class – will run before the resource on the right side. The syntax `<| |>` – also known as the "spaceshift" operator – is a **resource collector.** The collector represents a group of resources and consists of: a resource type, the `<|` operator, an optional search expression, and the `|>` operator. Since we left a blank search expression, it will select all resources of type `package`.

Putting these two pieces of information together, we have a more generic code that ensures Puppet will run `apt-get update` before trying to install

any package on the system. With that, we can change our `online_store` class, declared in the `modules/online_store/manifests/init.pp` file:

```
class online_store {
  class { 'apt':
    always_apt_update => true,
  }

  Class['apt'] -> Package <| |>
}
```

We can now remove references to the `Exec[apt-update]` resource from the `mysql` and `tomcat` modules, as well as remove the resource itself. The `modules/mysql/manifests/client.pp` file gets even more simplified:

```
class mysql::client {
  package { "mysql-client":
    ensure  => installed,
  }
}
```

Similarly, the `modules/mysql/manifests/server.pp` file also becomes shorter:

```
class mysql::server {
  package { "mysql-server":
    ensure  => installed,
  }

  file { "/etc/mysql/conf.d/allow_external.cnf": ... }
  service { "mysql": ... }
  exec { "remove-anonymous-user": ... }
}
```

Finally, we remove the unwanted dependency from the `tomcat::server` class, changing the `modules/tomcat/manifests/server.pp` file:

```
class tomcat::server($connectors = [], $data_sources = []) {
  package { "tomcat7":
    ensure  => installed,
  }

  file { "/etc/default/tomcat7": ... }
  file { "/var/lib/tomcat7/conf/server.xml": ... }
  file { "/var/lib/tomcat7/conf/context.xml": ... }
  service { "tomcat7": ... }
}
```

To complete this refactoring, we need to ensure that the code we added to the `online_store` class runs when provisioning the `web` and `db` servers. To achieve this, we simply include it in the beginning of the `online_store::web` and `online_store::db` class declaration:

```
# Contents of file modules/online_store/manifests/web.pp:
class online_store::web {
  include online_store
  ...
}

# Contents of file modules/online_store/manifests/db.pp:
class online_store::db {
  include online_store
  ...
}
```

Now our refactoring is complete. Refactoring changes the internal structure of the code without changing its external behavior [8]. To ensure that our servers are still configured correctly after all the changes we made in this chapter, we are going provision them from scratch, first destroying the `db` server:

```
$ vagrant destroy db
    db: Are you sure you want to destroy the 'db' VM? [y/N] y
==> db: Forcing shutdown of VM...
==> db: Destroying VM and associated drives...
==> db: Running cleanup tasks for 'puppet' provisioner...
```

However, when trying to bring the `db` server up again, we encounter an unexpected problem:

```
$ vagrant up db
Bringing machine 'db' up with 'virtualbox' provider...
...
Puppet::Parser::AST::Resource failed with error ArgumentError:
Could not find declared class apt at /tmp/vagrant-puppet-2/
modules-0/loja_virtual/manifests/init.pp:4 on node precise32
```

This error indicates that Puppet could not find the `apt` class, because we have not installed the PuppetLabs module. The simplest way to install a community module is by using Puppet's own command line tool, and run `puppet module install <module-name>`. This will install the new module and its dependencies in the default `modulepath`, making them available in the next execution of a `puppet apply`.

In our current setup, we do not run `puppet apply` directly. We are using Vagrant to manage Puppet's execution. Moreover, the module needs to be installed inside the virtual machine, which makes the provisioning process a bit more complicated if we have to run another command before Vagrant executes Puppet.

As the number of modules and their dependencies increase, it becomes more difficult to install them one by one from the command line. The **Librarian Puppet** library (http://librarian-puppet.com/) was created precisely to solve this problem. Once you declare all dependencies in a single `Puppetfile` file, Librarian Puppet resolves and installs the modules in the specified versions on an isolated directory. It is similar to the way Bundler works, for those familiar with the Ruby ecosystem.

At the same level where the `Vagrantfile` file is defined, we will create a new `librarian` directory containing the `Puppetfile` file with the following contents:

```
forge "http://forge.puppetlabs.com"

mod "puppetlabs/apt", "1.4.0"
```

The first line states that we will use PuppetLabs' Forge as the official source to resolve and download modules. The second line declares a dependency on the `pupppetlabs/apt` module, on version `1.4.0`.

To execute Librarian Puppet before running Puppet, we will install a new Vagrant plugin. Plugins are a way to extend Vagrant's basic behavior, and allow the community to write and share several improvements. Other examples of Vagrant plugins include support for other virtualization tools, virtual machines in the cloud, etc. To install the Librarian Puppet plugin (https://github.com/mhahn/vagrant-librarian-puppet/) , simply run:

```
$ vagrant plugin install vagrant-librarian-puppet
Installing the 'vagrant-librarian-puppet' plugin. This can
  take a few minutes...
Installed the plugin 'vagrant-librarian-puppet (0.6.0)'!
```

Since we are managing the Librarian Puppet modules in an isolated directory, we need to add it to the `modulepath`, changing our `Vagrantfile`:

```
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  ...
  config.librarian_puppet.puppetfile_dir = "librarian"

  config.vm.define :db do |db_config|
    ...
    db_config.vm.provision "puppet" do |puppet|
      puppet.module_path = ["modules", "librarian/modules"]
      ...
    end
  end

  config.vm.define :web do |web_config|
    ...
    web_config.vm.provision "puppet" do |puppet|
      puppet.module_path = ["modules", "librarian/modules"]
      ...
    end
```

```
  end
  ...
end
```

Having done that, we can try to provision the `db` VM again and, this time, if all goes well, we will see that our infrastructure automation code continues to work even after all the refactorings we have made so far:

```
$ vagrant reload db
==> db: Checking if box 'hashicorp/precise32' is up to date...
...
$ vagrant provision db
==> db: Installing Puppet modules with Librarian-Puppet...
==> db: Running provisioner: puppet...
Running Puppet with db.pp...
...
notice: Finished catalog run in 56.78 seconds
```

5.6 Conclusion

Mission accomplished! Compared with the beginning of the chapter, we have not only improved the structure and the organization of our code, but also learned new functionalities and features of Puppet. The files that were large and confusing each have a single line of code at the end of the chapter. Now, the manifest files represent what will be installed on each server. We also learned how to reuse community modules and how to manage dependencies using open source tools like Librarian Puppet and the respective Vagrant plugin.

As an exercise to reinforce your understanding, imagine a scenario where we wanted to install the entire online store on a single server: How difficult would this task be, now that we have well-defined modules? Another interesting exercise is to use what we have learned about the Puppet to automate the installation and configuration of the monitoring server from chapter 3.

But of course there is still room for improvement. Writing clean and maintainable code is harder than simply writing code that works. We need to reflect and decide when the code is good enough to move on.

For now, embracing the spirit of continuous improvement, we decided to stop refactoring even though we know there are still a few bad smells in our code. In particular, delivering a static WAR file as part of a Puppet module only works if you never have to change it. We know that this is not true in most situations, so we will revisit this decision in the next few chapters. But before that, we need to learn a safe and reliable way to generate a new WAR file every time we make a change to the online store code.

Chapter 6

Continuous integration

If you are a software developer, you might have learned a bit about the operations world in previous chapters. We provisioned the production environment, we built and deployed our application, we set up a monitoring system and we used automation with Puppet to treat infrastructure as code. We left aside everything that happens before the application is packaged into a WAR file to focus on the operational side's concerns when trying to put a system in production.

The essence of DevOps is to improve the collaboration between development and operation teams, so it is time to turn our attention to the development side. Writing code is one of the main activities of a software developer and DevOps can help make the delivery process more reliable. We will discuss how to deal with source code changes in a safe way, starting with a commit until the point where we have a package that can be deployed in production.

6.1 Agile engineering practices

Many companies start their Agile adoption using Scrum [16] or Kanban [3], processes that focus on management practices such as: stand-up meetings, card wall, using stories to divvy up work, planning on iterations or retrospectives. However, experienced Agile practitioners know that a successful adoption cannot ignore the Agile engineering practices that directly affects the quality of the software being developed.

Extreme Programming – or simply XP – was one of the first Agile methodologies that revolutionized the way software was developed in the late 90s. In the seminal book "Extreme Programming Explained" [5], Kent Beck outlines the values, principles and practices of the XP methodology, the main objective of which is to allow the development of high quality software that adapts to change its requirements.

The engineering practices introduced by XP include: refactoring, TDD, collective code ownership, incremental design, pair programming, coding standards and continuous integration. While some practices are still controversial, many of them can now be considered standard these days, and are being adopted by a large number of companies and teams. In particular, the practice of continuous integration was very successful and helped solve a big problem with the existing software development processes: the late integration phase.

We have already discussed in chapter 1 what happens when companies only integrate and test later, in the "last mile". So, it is important to dedicate a DevOps book chapter to the practice of continuous integration. However, before we define what it is and how to implement it, we must talk about some of its basic elements to understand why it works.

6.2 Starting with the basics: version control

Whether working alone or in a team, the first sign of a good and disciplined programmer is how they manage changes to the system's source code. When I was learning to program, I wrote throw-away code and simply kept a copy of all the files on my computer. This worked until the first time I had to revert the code to a previous stable version. I had made so many changes at once

that the code was inconsistent— it did not compile and did not work. All I wanted was a time machine to bring me to the last version that had worked.

When I started working in a team, this problem got worse. My colleagues and I wanted to make changes in different parts of the system, but since each one of us had a full copy of the code, we had to share directories, send email attachments or use floppy disks (yes, at that time this was common) to transfer files to each other. Needless to say, this was an arduous task and we made sporadic mistakes, which took us a long time to figure out and how to resolve conflicts.

My life changed when I discovered that this problem was already solved. **Version control systems** are tools that act as a centralized code repository that keeps a history of all the changes made to your code. Each change set is called a *commit*, which groups the files that changed, the author of the change, and a message with further explanation. This allows you to navigate the project history and know who changed what, when, and why. When two people try to change the same file, these tools try to resolve the conflict automatically, but if needed they will request human help.

Nowadays there is no reason not to use a version control system. One of the most popular tools in this space is **Git** (http://git-scm.com/) , a distributed system that does not require a single central repository for all project commits. Since it is distributed, each developer can run a local server on their own machine. What makes it a powerful tool is that each server knows how to send individual local histories to other servers, allowing people to share code in an efficient manner. The choice of a centralized server in Git is only a convention, but no one needs network connectivity to make local commits.

It still amazes me how there are teams that do not have any discipline in their delivery process. Developers simply login on the production server and change the code "live" without any form of audit or quality control. Any developer can introduce a defect in production and nobody will know the source of the problem.

Continuous Delivery and DevOps require discipline and collaboration. The delivery process of a disciplined team **always** starts with a commit. Any change that needs to go to production – source code, infrastructure code or configuration – starts with a commit on a version control system. This lets

you track potential problems in production with its origin and its author, in addition to serving as a central tool for collaboration between development and operation teams.

The online store is using Git as a version control system. Its main repository is hosted on Github. Besides offering free Git repositories for open source projects, Github is a collaboration tool that allows other users to report issues, request new features, or even clone repositories to contribute with bug fixes or implement new features.

In order to follow the examples in this chapter, it is recommended that you have your own Git repository for the online store, where you will have the freedom to make your own commits. To do that, you can simply create your own fork of the main repository on GitHub. A **fork** is nothing more than your own clone of any GitHub repository. This functionality exists so that members of the open source community can make modifications and contribute them back to the main repository.

To create your fork you will need a registered user on GitHub. After that, go to the online store repository URL at https://github.com/dtsato/loja-virtual-devops and click the *Fork* button, as shown in figure 6.1. After a few seconds, you will have a complete copy of the repository on a new URL that is similar to the previous URL, but replaces the user `dtsato` with your GitHub user.

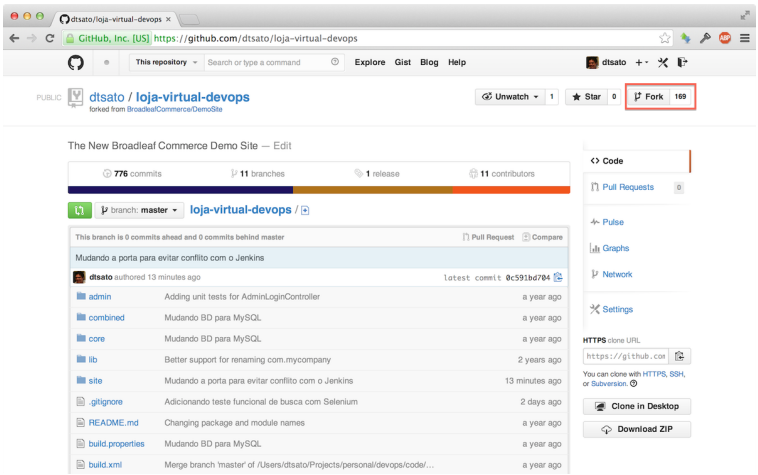

Fig. 6.1: Forking a GitHub repository

Once your repository has been forked, you can run the following command to get a local copy in your machine, replacing `<username>` with your GitHub user:

```
$ git clone https://github.com/<username>/loja-virtual-devops.git
Cloning into 'loja-virtual-devops'...
remote: Counting objects: 11358, done.
remote: Compressing objects: 100% (3543/3543), done.
remote: Total 11358 (delta 6053), reused 11358 (delta 6053)
Receiving objects: 100% (11358/11358), 55.47 MiB | 857.00 KiB/s,
                    done.
Resolving deltas: 100% (6053/6053), done.
$ cd loja-virtual-devops
```

In any Git repository, you can use the `git log` command to show the history of the project's commits, for example:

```
$ git log
commit 337865fa21a30fb8b36337cac98021647e03ddc2
Author: Danilo Sato <dtsato@gmail.com>
Date:   Tue Mar 11 14:31:22 2014 -0300
```

```
    Adding LICENSE

commit 64cff6b54816fe08f14fb3721c04fe7ee47d43c0
Author: Danilo Sato <dtsato@gmail.com>
...
```

Each commit in Git is represented by a *hash* – or SHA – which is a series of letters and numbers that act as a unique identity. In the previous example, the most recent commit has a SHA starting with `337865da`. The SHA is calculated based on the contents of the files modified in the commit and it works as a kind of primary key that can be used to reference that commit.

In other version control systems, such as SVN, commits are represented by an auto-incrementing number for each change. However, since Git is a distributed system, each repository needs to calculate a hash which is globally unique, regardless of the existence of other repositories.

Later on we will see how to make new commits in the local repository and how to share them with the central repository. For now, we will learn other tools and Agile engineering practices that facilitate the adoption of DevOps and Continuous Delivery.

6.3 Automating the build process

We have seen the power of automation in previous chapters when we replaced a manual provisioning and deployment process with Puppet code. The automation of repetitive tasks helps decrease the risk of human errors, as well as shortening the feedback loop between introducing a problem and detecting it. Automation is one of the main pillars of DevOps and it is highlighted by several Agile engineering practices.

A part of the delivery process which greatly benefits from automation is the **build process.** The build of a system involves all the tasks required to run it, such as: downloading and resolving dependencies, compilation, library linking, packaging, obtaining source code quality metrics, etc. These tasks may vary depending on the programming language you use. In Ruby, for example, you do not need to compile files before executing them.

The result of a build is usually one or more binary files, also known as **artifacts.** In a Java system, a few examples of artifacts are: `.jar` and `.war` files,

Javadoc documentation in HTML produced from source code, automated test reports or those generated by tools that calculate source code quality metrics.

An example of a non-software related build process is being used while writing of this book. The book is written in text files using a markup language developed by Caelum – **Tubaína** (https://github.com/caelum/tubaina) . Our build process transforms these text files and images into artifacts that can be printed or distributed as e-books in PDF format, `.mobi` or `.epub`.

As a system grows, its build process gets more complicated and it becomes difficult to remember all the necessary steps. That's why it is is important to invest in automation. Currently, all major programming languages have one or more open source build tools available to assist in the build process: in Java we have Ant, Maven, Buildr, Gradle; in Ruby we have Rake; in Javascript we have Grunt and Gulp; in .NET we have NAnt; besidesas well as many other alternatives.

The online store has been written using Maven as its build tool. As discussed in the end of chapter 2, our build process will resolve and download all library dependencies, compile, run automated tests, and package artifacts for 4 distinct modules: `core`, `admin`, `site` and `combined`. To make sure that the Maven build still works on your clone of the repository, you can run the command:

```
$ mvn install
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------
...
[INFO] ------------------------------------------------------
[INFO] Reactor Summary:
[INFO] ------------------------------------------------------
[INFO] loja-virtual-devops ..............  SUCCESS [1.156s]
[INFO] core .............................  SUCCESS [2.079s]
[INFO] admin ............................  SUCCESS [10:16.469s]
[INFO] site .............................  SUCCESS [7.928s]
[INFO] combined .........................  SUCCESS [25.543s]
[INFO] ------------------------------------------------------
[INFO] ------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------
```

```
[INFO] Total time: 10 minutes 53 seconds
[INFO] Finished at: Thu Mar 14 07:15:24 UTC 2013
[INFO] Final Memory: 80M/179M
[INFO] ------------------------------------------------
```

For now, we will not make any changes to the build process, but there is an important aspect of the process that deserves more attention: the automated tests.

6.4 Automated testing: reducing risk and increasing confidence

Compiling the code is an important step for its execution, especially in statically typed languages. However, it does not guaranteed that the system will work as you expect. It can contain defects or even have wrongly implemented features. So, it is also important to include in **automated tests** as part of the build process.

There are several types of tests that can be automated, each with distinct characteristics. There is no widely accepted classification or terminology for the different types of tests in our industry, but some of the most common are:

- **Unit tests**: Practices like TDD [4] and refactoring [8] put a lot of emphasis on this type of testing, which exercises units of code in isolation without any need to spin up the entire application. These tests usually run pretty fast – in the order of milliseconds – and can be executed frequently to provide the best kind of feedback for developers.
- **Integration tests** or **component tests**: These tests exercise a part of the application or how it integrates with its dependencies; for example, a database, a REST service, a message queue, or even with its frameworks and libraries. Depending on the component that is being tested, executing these tests may require parts of the system to be running.
- **Functional tests** or **acceptance tests**: These tests exercise the system as a whole, from an external point of view. Generally, this type of test

exercises the user interface or simulates a user interacting with the system. These tests are generally slower because they require most of the system to be running.

And these are not the only types of tests—there are many more: performance tests, exploratory tests, usability tests, etc. They all contribute to an assessment of the quality of your system, but not all of them are easily automated. Exploratory tests, for example, are manual by nature and require the creativity of a human to figure out scenarios where the system does not behave as expected.

The Agile community generally refers to the four quadrants shown in figure 6.2, created by Brian Marick [7] as a model to describe the types of test required to deliver quality software.

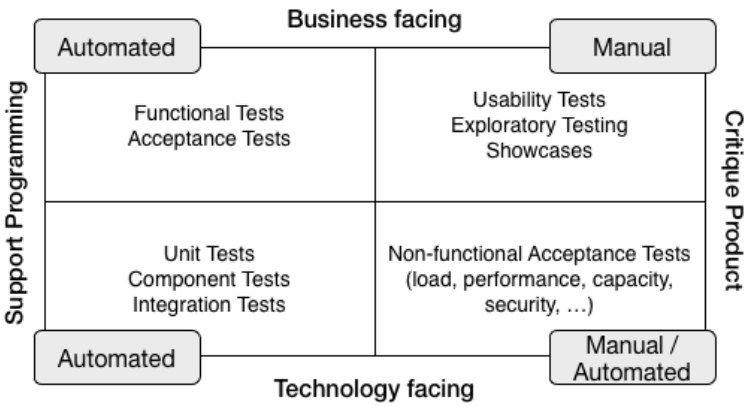

Fig. 6.2: Testing quadrant

On the left side are tests that support programming, or tests that developers can run fairly often to get feedback about the changes introduced in each commit. These tests assess the internal quality of the software.

On the right side are tests that critique the product as a whole, assessing its external quality and trying to find noticeable problems for someone using the system.

On the upper side are business facing tests. These tests are described in terms of the business, using a language that makes sense to a domain expert.

On the lower side are technology facing tests. These tests are described in technical terms, using a language that makes more sense to developers but not that much to a domain expert.

The diagram also shows examples of the types of tests for each quadrant as well as places where it is more common to invest in automation. Tests in the upper right quadrant are usually manual, because they require creativity and exploration— tasks that humans are good at doing. Tests in the lower left quadrant are usually automated in order to be executed several times and prevent regression issues. In this diagonal, there is not much controversy in the industry and many companies adopt this testing strategy. The other diagonal is a little more controversial.

A major problem in the software industry is how companies look at the tests in the upper left quadrant. It is very common to see companies approach these tests manually and invest a huge amount of time and people to write test scripts on a spreadsheet or an elaborate test case management system.

We humans are terrible at executing these step by step scripts because we are not good at repetitive tasks: it is common for us to skip a step or even execute them in the wrong order. For this reason there are still companies allocating a considerable amount of time in the "last mile" for a phase of comprehensive testing. An Agile approach for the tests on the upper left quadrant is to invest in automation, so humans can be freed up to run tests on the upper right quadrant, where we can add a lot more value.

In the lower right quadrant, the problem is slightly different. There are ways to automate some types of tests in this quadrant, while others still have to be manual. However, the most common problem is that this quadrant is usually ignored. Performance problems or scalability are only found in production, when it is too late. This is an area where the collaboration brought by DevOps can bring enormous benefit to the delivery of quality software. In the case of our online store, since a great part of the system is implemented by *Broadleaf Commerce*, we do not have a lot of custom Java code, and therefore few automated tests. The original demo store had no tests at all, although the framework code itself is well tested. We could write another book just on this topic, but we decided to automate just a few unit and functional tests to use as examples.

Our main interest is to use these tests in order to provide feedback when changes are introduced. The longer it takes between the moment a defect is introduced and its detection, the harder it is to fix it. For this reason, it is important to include the largest possible number of automated tests in the build process.

Maven expects you to write unit tests and it knows how to execute them in the default build process. If you examine the output of the `mvn install` command that we ran in the previous section, you will see the unit test results as part of the build for the `admin` module:

```
...
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running br.com.devopsnapratica.....AdminLoginControllerTest
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time ...

Results :

Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
...
```

Our Maven build is also setup to run functional tests that we will write using **Selenium** (https://code.google.com/p/selenium/) , currently also known as **WebDriver.** Selenium is a library for automating web browsers, which enables us to simulate the behavior of a user navigating the online store and write automated tests in JUnit style to verify that everything is working as expected.

In the `site` module, we created just a single example of a functional test defined in the `br.com.devopsnapratica.acceptance.SearchTest` class. Initially, we need to choose which browser to use and what URL to access:

```
@Before
public void openBrowser() throws InterruptedException {
    driver = new HtmlUnitDriver();
    driver.get("http://localhost:8080/");
    Thread.sleep(5000); // Waiting for Solr index
}
```

The `@Before` annotation is part of JUnit and it means that the method will be executed before any test. In this case, we create an instance of an `HtmlUnitDriver` which is the fastest and lightest implementation of a `WebDriver` because, instead of opening a new process and launching a new browser window, it emulates everything in Java. Other implementations exist for the most common browsers, such as: `ChromeDriver`, `FirefoxDriver` or, if you are on Windows, `InternetExplorerDriver`.

The `get(String url)` method loads a web page at the specified URL, blocking it until the operation completes. Finally, we make our test wait 5 seconds to ensure that the Solr search index has been loaded. Our test method is defined using JUnit's `@Test` annotation:

```
@Test
public void searchScenario() throws IOException {
  ...
}
```

First, we verify that the page was loaded correctly by checking the browser window title using the `getTitle()` method from the `driver`:

```
assertThat(driver.getTitle(),
           is("Broadleaf Demo - Heat Clinic"));
```

Knowing that we are on the correct page, we search for the element corresponding to the search field, named `q`, using the `findElement(By by)` method. We then fill out the field with the search term *"hot sauce"* and submit the form using the `submit()` method:

```
WebElement searchField = driver.findElement(By.name("q"));
searchField.sendKeys("hot sauce");
searchField.submit();
```

The `submit()` method will also block until the new page is loaded. Once this is done, just verify that the search has returned 18 products, validating the description text with the `getText()` method:

```
WebElement searchResults = driver.findElement(
           By.cssSelector("#left_column > header > h1"));
assertThat(searchResults.getText(),
          is("Search Results hot sauce (1 - 15 of 18)"));
```

Notice that this time we are using a CSS selector to find the desired element on the page. Finally, we can also verify that the current page only shows 15 results, now using the `findElements(By by)` method which, instead of returning a single element, returns a list of elements:

```
List<WebElement> results = driver.findElements(
            By.cssSelector("ul#products > li"));
assertThat(results.size(), is(15));
```

The Maven build is setup to execute our functional test as part of the `integration-test` phase, because we need to startup both the database and the the application before running this type of test. Looking at the output of the `mvn install` command executed previously, you will also find the functional test results in the build of the `site` module:

```
...
[INFO] --- maven-surefire-plugin:2.10:test (integration-test)
           @ site ---
[INFO] Surefire report directory: ...

-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running br.com.devopsnapratica.acceptance.SearchTest
...
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time ...

Results :

Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
...
```

With this, we have shown how to integrate different types of automated tests in the build process of your project. Now we have the basic ingredients to discuss an essential practice to develop on a team in a safe and efficient manner: **continuous integration.**

6.5 What is continuous integration?

When multiple developers are working on the same system, new challenges emerge in the software development process. To work in parallel requires more communication and coordination between team members, because the longer they work without integrating their changes, the greater the risk of having conflicts.

Continuous Integration is one of the original XP practices that encourages developers to integrate their work frequently so that potential problems are detected and fixed quickly. When developers finish a task whose code can be shared with the team, they must follow a disciplined process to ensure that their changes will not introduce problems and these changes will work properly with the rest of the code.

When you finish your task, you execute a local build that will run all the automated tests. This ensures that your changes work in isolation and can be integrated with the rest of the code. The next step is to update your local copy of the project, pulling the latest changes from the central repository. You must then run the tests one more time to ensure that your changes work with other changes that have been introduced by other developers. Only then can you push your commit to the central repository.

Ideally, each developer should commit at least once a day to avoid accumulating local changes that are not visible to the rest of the team. Mature teams have the practice of committings even more often, usually several times per day. In the worst case, if you are working on a change that will take longer and there is no way to break it into smaller pieces, you must update your local copy of the project as much as possible to avoid diverging from what is happening in the central repository.

This ritual reduces the chance of your introducing a change that will break the project's tests. However, it also requires that every developer has the discipline to follow this process on every commit. Therefore, although not strictly necessary, it is very common for teams to provision a server dedicated to performing continuous integration, which will be responsible for ensuring the process is being followed correctly.

The continuous integration server is responsible for monitoring the central code repository and triggering a project build every time it detects a new

commit. At the end of the build execution, it will pass or fail. When the build fails, it is common to say that "the build is broken" or that "it is red." When the build finishes successfully, it is common to say that "the build passed" or that "it is green."

The continuous integration server also acts as an information radiator for the team, keeping all developers informed of the project's current build status. Tools in this space have several ways to disseminate that information: through dashboards and web interfaces, sending emails or notifications to the team every time the build fails, or exposing an API that can be used to show the current build status on other devices.

When an integration problem is detected and the build breaks, a good practice is that no one share commits until the build is fixed. Generally, the developer responsible for the last commit begins to investigate the problem and decides between fixing the problem or reverting the problematic commit. This is a good policy for highlighting the importance of having a green build, since the team is prevented from sharing new changes while the problem is not resolved.

There are several commercial and open source tools that can be used as a continuous integration server. Some of the most popular are **Jenkins** (http://jenkins-ci.org/) , **CruiseControl** (http://cruisecontrol.sourceforge.net/) , **Go** (http://go.thoughtworks.com/) , **TeamCity** (http://www.jetbrains.com/teamcity/) and **Bamboo** (http://www.atlassian.com/software/bamboo) . There are also cloud continuous integration solutions offered as SaaS (*Software as a Service*) such as **TravisCI** (http://travis-ci.org/) , **SnapCI** (http://snap-ci.com/) or **Cloudbees** (http://cloudbees.com/) . For a more complete list of the different continuous integration tools, it is worth visiting the Wikipedia page http://en.wikipedia.org/wiki/Comparison_of_continuous_integration_software.

Our goal is not to go into the advantages and disadvantages of each tool, but to show an example of a continuous integration process working in practice. That is why we chose to use Jenkins, which is a very popular open source tool in the Java community, with a broad plugin ecosystem and a very active community.

6.6 Provisioning a continuous integration server

Since we are already familiar with the provisioning process using Vagrant and Puppet, we will use them to create a new virtual machine called `ci` on our `Vagrantfile` configuration:

```
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  ...
  config.vm.define :db ...
  config.vm.define :web ...
  config.vm.define :monitor ...

  config.vm.define :ci do |build_config|
    build_config.vm.hostname = "ci"
    build_config.vm.network :private_network,
                            :ip => "192.168.33.16"
    build_config.vm.provision "puppet" do |puppet|
      puppet.module_path = ["modules", "librarian/modules"]
      puppet.manifest_file = "ci.pp"
    end
  end
end
```

The new VM will be provisioned with Puppet using a new manifest file called `ci.pp`. You can create this new file in the `manifests` directory—its contents are quite simple:

```
include online_store::ci
```

Following the same pattern from chapter 5, we will define the CI server resources in a new `online_store::ci` class inside the `online_store` module. In order to do that, we need to create a new file in the path `modules/online_store/manifests/ci.pp` in which this new class is defined:

```
class online_store::ci {
  include online_store
```

```
  ...
}
```

This is when we get to the interesting part of installing and configuring Jenkins. First, we need to install some basic development and build tool packages such as Git, Maven and the Java development kit:

```
package { ['git', 'maven2', 'openjdk-6-jdk']:
  ensure => "installed",
}
```

To install and setup Jenkins, we will use a module from Puppet Forge, whose documentation can be found in https://github.com/jenkinsci/puppet-jenkins. This module defines a class called `jenkins` that installs a Jenkins server with default settings. To use it, simply add the resource:

```
class { 'jenkins':
  config_hash => {
    'JAVA_ARGS' => { 'value' => '-Xmx256m' }
  },
}
```

We are passing a configuration hash that will override the default setting for one of Jenkins' initialization options. Defining the `JAVA_ARGS` setting with the value `"-Xmx256m"` defines that the Jenkins Java process can allocate and use up to 256Mb of memory.

One of Jenkins' main attractions is its plugin ecosystem. For a complete list of all Jenkins supported plugins you can access their documentation at https://wiki.jenkins-ci.org/display/JENKINS/Plugins. The Puppet module has a defined type that allows us to install several plugins:

```
$plugins = [
  'ssh-credentials',
  'credentials',
  'scm-api',
  'git-client',
  'git',
  'javadoc',
```

```
  'mailer',
  'maven-plugin',
  'greenballs',
  'ws-cleanup'
]

jenkins::plugin { $plugins: }
```

In our case, the main plugins that we are interested in are:

- **`git` plugin** (http://wiki.jenkins-ci.org/display/JENKINS/Git+Plugin) : Allows Jenkins to monitor and trigger builds from a Git repository instead of the default SVN option. The `ssh-credentials`, `credentials`, `scm-api` and `git-client` plugins are just dependencies of that plugin, which is why we need to include them.
- **`maven-plugin` plugin** (http://wiki.jenkins-ci.org/display/JENKINS/Maven+Project+Plugin) : Allows Jenkins to run a Maven build. The `javadoc` and `mailer` plugins are just dependencies of that plugin, which is why we need to include them.
- **`greenballs` plugin** (http://wiki.jenkins-ci.org/display/JENKINS/Green+Balls) : Makes a cosmetic change to Jenkins' web interface to show successful builds in green color instead of the default blue.
- **`ws-cleanup` plugin** (https://wiki.jenkins-ci.org/display/JENKINS/Workspace+Cleanup+Plugin) : Allows Jenkins to cleanup files between consecutive build executions.

With that, the entire `online_store::ci` class should look like:

```
class online_store::ci {
  include online_store

  package { ['git', 'maven2', 'openjdk-6-jdk']:
    ensure => "installed",
  }
```

```
  class { 'jenkins':
    config_hash => {
      'JAVA_ARGS' => { 'value' => '-Xmx256m' }
    },
  }

  $plugins = [
    'ssh-credentials',
    'credentials',
    'scm-api',
    'git-client',
    'git',
    'maven-plugin',
    'javadoc',
    'mailer',
    'greenballs',
    'ws-cleanup'
  ]

  jenkins::plugin { $plugins: }
}
```

We may now try to bring the `ci` server up using Vagrant, but we have found a problem:

```
$ vagrant up ci
Bringing machine 'ci' up with 'virtualbox' provider...
==> ci: Importing base box 'hashicorp/precise32'...
==> ci: Matching MAC address for NAT networking...
...
Puppet::Parser::AST::Resource failed with error ArgumentError:
Could not find declared class jenkins at
/tmp/vagrant-puppet-2/modules-0/online_store/manifests/ci.pp:12
on node precise32
```

The `jenkins` class was not found because we forgot to add it to the `Puppetfile` that Librarian Puppet uses to install Puppet Forge modules. For this, we need to change the `librarian/Puppetfile` file by adding the `rtyler/jenkins` module in the `1.0.0` version:

```
forge "http://forge.puppetlabs.com"

mod "puppetlabs/apt", "1.4.0"
mod "rtyler/jenkins", "1.0.0"
```

Now we may re-provision the `ci` server and Jenkins will be successfully installed:

```
$ vagrant provision ci
==> ci: Installing Puppet modules with Librarian-Puppet...
==> ci: Running provisioner: puppet...
Running Puppet with ci.pp...
...
notice: Finished catalog run in 84.19 seconds
```

To verify that Jenkins was properly installed and configured, you may open a new browser window and access the URL http://192.168.33.16:8080/. If everything is correct, you will see the Jenkins welcome screen shown in figure 6.3.

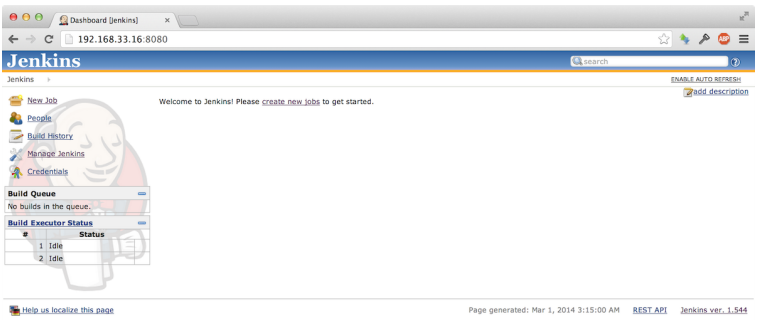

Fig. 6.3: Jenkins welcome screen

6.7 Configuring the online store build

Now that Jenkins is up and running, we need to configure it to run the online store every time there is a new commit in the Git repository. However, before doing so, we need to do a manual setup so that Jenkins recognizes where Maven is installed in the system.

For this, you must click the *"Manage Jenkins"* link on the home page, and on the next screen click the first *"Configure System"* link, which will take you to Jenkins' administration page shown in figure 6.4.

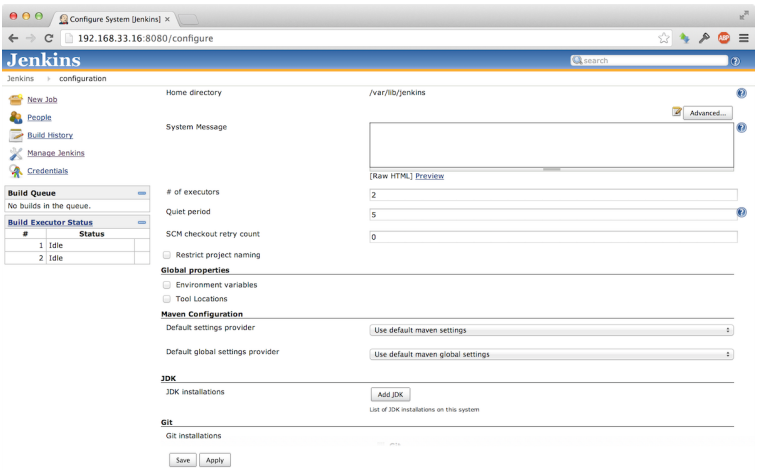

Fig. 6.4: Jenkins system configuration screen

In this screen, a little bit further down, you will see a *"Maven"* section with a *"Maven installations"* option. Click the *"Add Maven"* button and it will expand new options. We will then uncheck the *"Install automatically"* option because Maven is already installed on the system. We just need to tell Jenkins the path where Maven can be found, filling out the *"Name"* field with the *"Default"* value and the *"MAVEN_HOME"* field with the value *"/usr/share/maven2"*. Figure 6.5 shows how your configuration should look like after following these instructions, with the important areas highlighted in red.

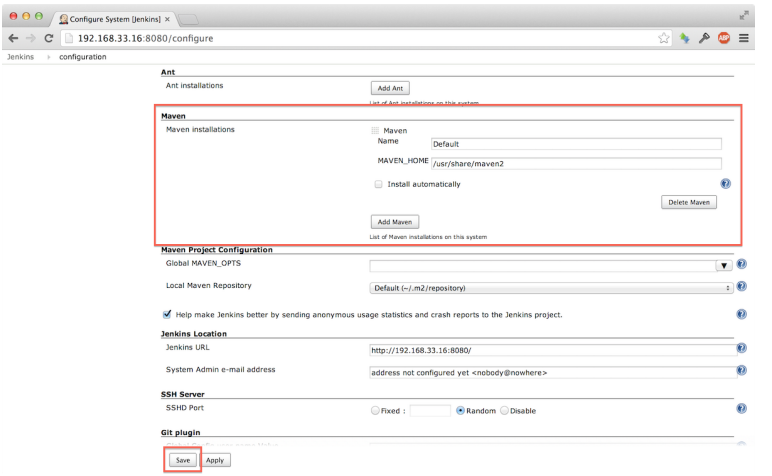

Fig. 6.5: Configuring Maven in Jenkins

Then you must click the *"Save"* button at the bottom of the page for the changes to take effect. This will take us back to Jenkins' welcome screen. We may then proceed to create the online store build by clicking the *"create new jobs"* link. In Jenkins terminology, a **job** represents the work you want to execute repeatedly. In the job, you set the execution parameters and task steps of your build. The job records the execution history as well as the current state of your build.

A new screen will appear for you to define the job name and its type. We will define the name as *"loja-virtual-devops"* and we will choose the *"Build a maven2/3 project"* option, as shown in figure 6.6.

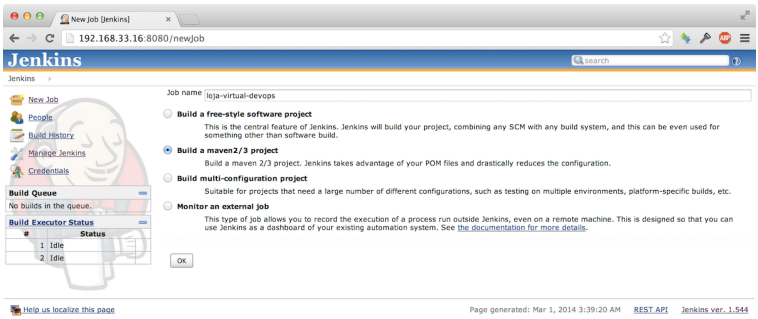

Fig. 6.6: Creating a new Jenkins job

Clicking the *"OK"* button we get to the main configuration screen for our new job. We will describe what needs to be done in each section, step by step:

- ***"Source Code Management"* section**: In this section we define which version control system Jenkins should use to get the project's code. By choosing *"Git"*, new options will be expanded. The only important option is *"Repository URL"* where you must fill in the URL of your GitHub repository, created at the end of the 6.2 section as shown in figure 6.7.

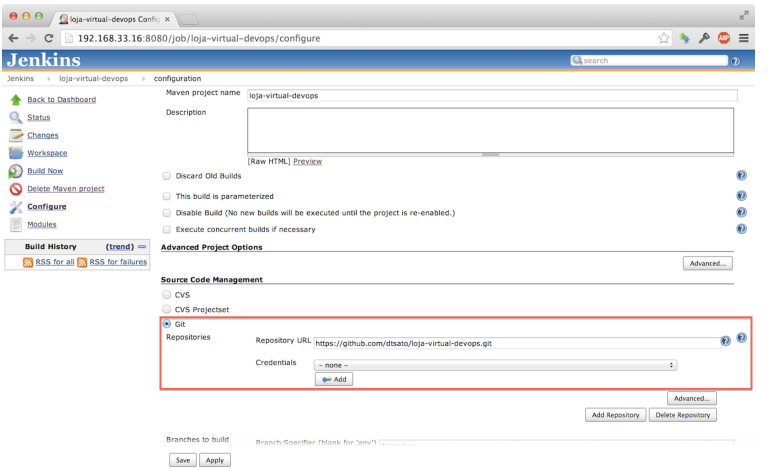

Fig. 6.7: Configuring the source control management

- ***"Build Triggers"* section**: In this section we define which actions will trigger a new execution of the project build. We will uncheck the *"Build whenever a SNAPSHOT dependency is built"* option and choose the *"Poll SCM"* option, which will monitor the version control system. A new *"Schedule"* text area will expand, where you will enter the frequency in which Jenkins will poll the repository for new commits. We will fill it in with the value "* * * * *", which means "monitor changes every minute", as shown in figure 6.8

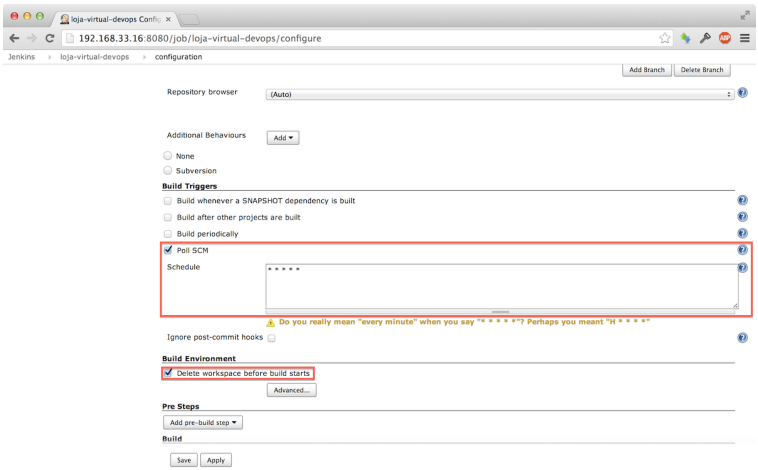

Fig. 6.8: Configuring job trigger and workspace cleanup

- ***"Build Environment"* section**: In this section we will choose the *"Delete workspace before build starts"* option so that Jenkins will cleanup the workspace before starting a new build, as shown in figure 6.8. This increases the build determinism, since no files will remain between build executions.

- ***"Build"* section**: In this section we will define which commands should be executed to run the build. As we are using the Maven plugin, we only need to fill out the *"Goals and options"* field with the value *"install"*, as shown in figure 6.9.

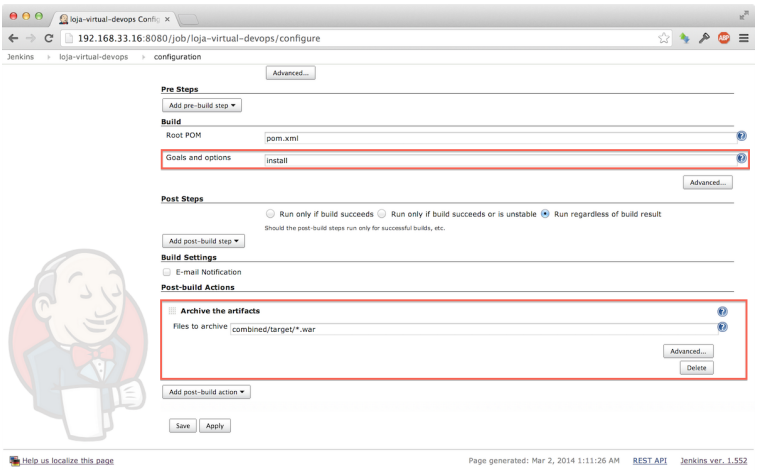

Fig. 6.9: Maven build goal and post-build action to archive artifacts

- ***"Post-build Actions"* section**: In this section we define what should happen after the build runs. By clicking the *"Add post-build action"* button, we choose the *"Archive the artifacts"* option, which tells Jenkins to archive the artifacts generated by the build process. New options will expand and you only have to fill out the *"Files to archive"* field with the value *"combined/target/*.war"*, as shown in figure 6.9. That will tell Jenkins to archive the `.war` artifact generated at the end of the build so we can use it later on, when we want to deploy it.

After completing all these settings, you can click on the *"Save"* button at the end of the page and the job will be created. Jenkins will start to monitor the Git repository immediately. In less than a minute you will see that your first build will be scheduled and, when build number 1 starts running, you will see a progress bar at the bottom right, as shown in figure 6.10.

Fig. 6.10: First build scheduled for execution

The first build usually takes a little longer because Maven has to resolve and download all the project dependencies. To follow the progress of the build and look at its output as if you were running it on a terminal, you may click on the progress bar. This page will refresh automatically and, at the end of the build execution you will see the message *"Finished: SUCCESS"* which tells us the build has passed! Figure 6.11 shows an example of a successful build.

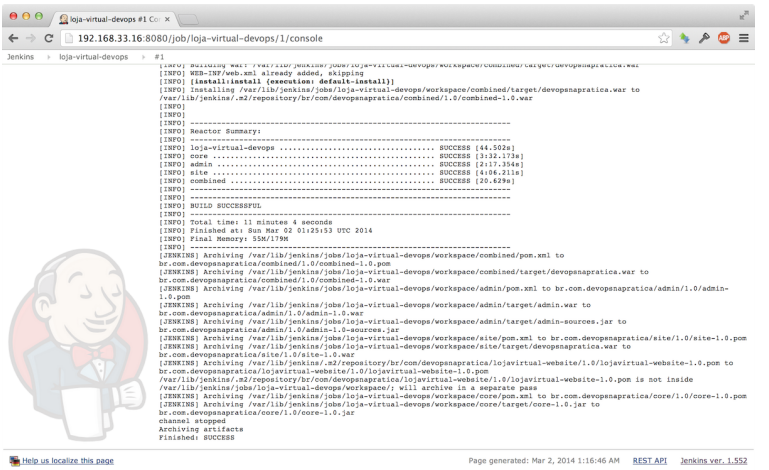

Fig. 6.11: Build successfully completed

For a summary of the build execution, go to the job URL in http://192.168.33.16:8080/job/loja-virtual-devops/ and you will see a screen similar to the one in figure 6.12. In the lower left corner you will also see the project's build history, with the green ball representing success – when the build fails, it will turn red – and at the center of the page you will see a link to the `.war` artifact, generated in the last successful build.

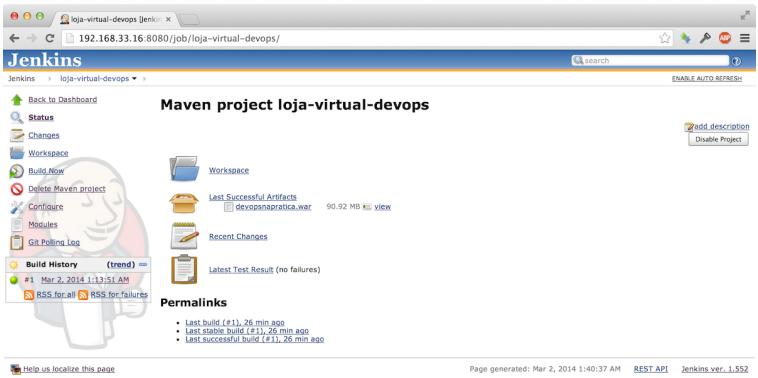

Fig. 6.12: Job overview and archived artifact

Now we have a continuous integration server that runs a build every time it detects a new commit in the Git repository. To test the installation further, let's make a new commit with a small change and validate that Jenkins will trigger a new project build. For this, we will use the repository that you cloned from GitHub in the preceding section. Within the cloned project's directory, run the following command:

```
$ echo -e "\n" >> README.md
```

The `echo` command will add a new line break at end of the `README.md`. file.

If you are following these examples on Windows and the command does not work, you can simply open the file in your favorite text editor, add the line break, and save it.

Next, the `git commit` command creates a new commit with a message explaining what has changed, and then the `git push` command will send the local commit to the central repository on GitHub. On this last command, you may need to provide your GitHub credentials, depending on how your account is configured.

```
$ git commit -am"Adding a line break at the end of README"
[master 4d191bd] Adding a line break at the end of README
 1 file changed, 2 insertions(+), 1 deletion(-)
$ git push origin master
Counting objects: 15, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 310 bytes | 0 bytes/s, done.
Total 3 (delta 2), reused 0 (delta 0)
To git@github.com:dtsato/loja-virtual-devops.git
   9ae96a8..4d191bd  master -> master
```

Once Jenkins detects the new commit, it will schedule a new build, as shown in figure 6.13.

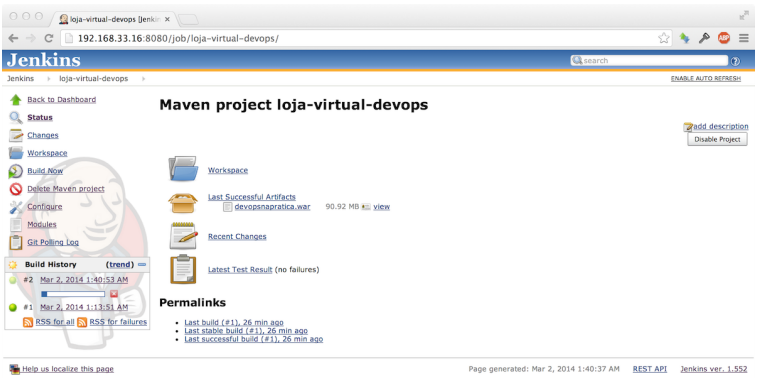

Fig. 6.13: New commit triggers a new build

This time, the build should execute faster. In our example, while the first build took about 12 minutes, the second took only 4 minutes because all dependencies were already fetched in Maven's local cache. At the end of the build, you may go back to the job overview page in http://192.168.33.16:8080/job/virtual-store-devops/ and see not only the latest available artifact but also a graph showing the execution history of the project's tests, as shown

in figure 6.14.

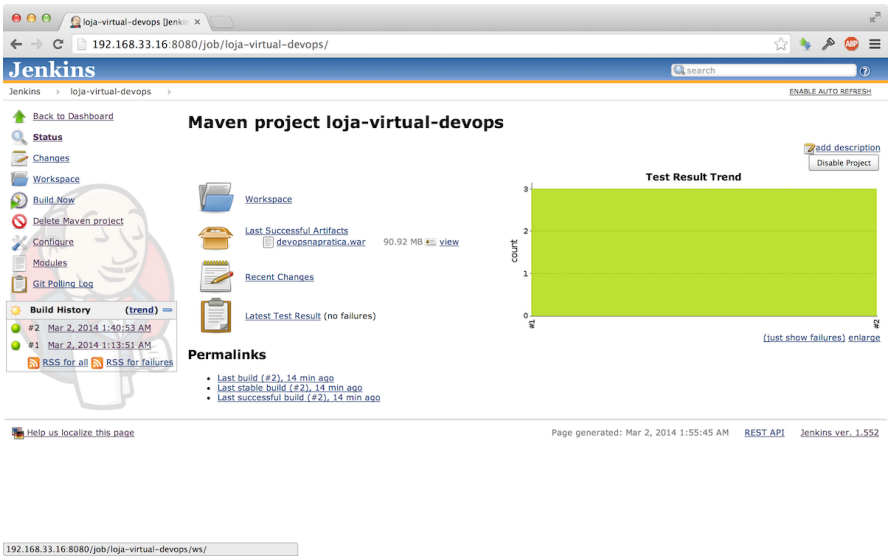

Fig. 6.14: Another successful build

6.8 Infrastructure as code for the continuous integration server

With Jenkins installed and working properly, we could end the chapter here. However, you may have noticed that we did several manual settings through Jenkins web UI and this can be dangerous if there are problems with our CI server.

When a team begins practicing continuous integration, that server becomes an essential component of the project's infrastructure. In the same way that no developer can commit when the build is broken, the whole team is blocked when the CI server is down.

For this reason, we need to be able to launch a completely new CI server as soon as possible. We will automate all those manual steps in Puppet code so we can provision a new CI server in a case of emergency.

Jenkins stores most of its internal settings in XML files in the directory

`/var/lib/jenkins`. The first manual setting we did in the last section was to tell Jenkins where Maven is installed on the system. This setting is saved in the `/var/lib/jenkins/hudson.tasks.Maven.xml` file. Let's use the same trick from chapter 4 to copy the file from inside the `ci` VM to our local machine:

```
$ vagrant ssh ci -- 'sudo cp \
> /var/lib/jenkins/hudson.tasks.Maven.xml \
> /vagrant/modules/online_store/files'
```

This command will save the file inside the `online_store` Puppet module, in `modules/online_store/files`. We need to add a new resource to the `online_store::ci` class by changing the `modules/online_store/manifests/ci.pp` file:

```
class online_store::ci inherits online_store {
  package { ... }
  class { 'jenkins': ... }
  $plugins = ...
  jenkins::plugin { ... }

  file { '/var/lib/jenkins/hudson.tasks.Maven.xml':
    mode    => 0644,
    owner   => 'jenkins',
    group   => 'jenkins',
    source  =>
     'puppet:///modules/online_store/hudson.tasks.Maven.xml',
    require => Class['jenkins::package'],
    notify  => Service['jenkins'],
  }
}
```

The next manual configuration we did was to create the job. Our job configuration was saved in the `/var/lib/jenkins/jobs/loja-virtual-devops/config.xml` file. Let's use the same trick to copy this file from the VM to a new `templates` directory in the `online_store` Puppet module:

```
$ mkdir modules/online_store/templates
$ vagrant ssh ci -- 'sudo cp \
> /var/lib/jenkins/jobs/loja-virtual-devops/config.xml \
> /vagrant/modules/online_store/templates'
```

To instruct Puppet on how to create the job after installing Jenkins, we need to define some variables on the `ci.pp` file. Do not forget to replace the username `dtsato` with your GitHub user in the URL of the `$git_repository` variable:

```
class online_store::ci inherits online_store {
  package { ... }
  class { 'jenkins': ... }
  $plugins = ...
  jenkins::plugin { ... }
  file { '/var/lib/jenkins/hudson.tasks.Maven.xml': ... }

  $job_structure = [
    '/var/lib/jenkins/jobs/',
    '/var/lib/jenkins/jobs/loja-virtual-devops'
  ]
  $git_repository =
    'https://github.com/dtsato/loja-virtual-devops.git'
  $git_poll_interval = '* * * * *'
  $maven_goal = 'install'
  $archive_artifacts = 'combined/target/*.war'
}
```

We must then replace the values that are hard-coded in the XML file with ERB expressions that use the variables we have just created. The XML elements that must be changed in the `modules/online_store/templates/config.xml` file are highlighted next, while the unchanged portions are represented by an ellipsis (...):

```
<?xml version='1.0' encoding='UTF-8'?>
<maven2-moduleset plugin="maven-plugin@2.1">
  ...
  <scm class="hudson.plugins.git.GitSCM" plugin="git@2.0.3">
```

```
    ...
    <userRemoteConfigs>
      <hudson.plugins.git.UserRemoteConfig>
        <url><%= git_repository %></url>
      </hudson.plugins.git.UserRemoteConfig>
    </userRemoteConfigs>
    ...
  </scm>
  ...
  <triggers>
    <hudson.triggers.SCMTrigger>
      <spec><%= git_poll_interval %></spec>
      ...
    </hudson.triggers.SCMTrigger>
  </triggers>
  ...
  <goals><%= maven_goal %></goals>
  ...
  <publishers>
    <hudson.tasks.ArtifactArchiver>
      <artifacts><%= archive_artifacts %></artifacts>
      ...
    </hudson.tasks.ArtifactArchiver>
  </publishers>
  ...
</maven2-moduleset>
```

With the template created, we need to change the `online_store::ci` class to create a new directory for our job and use the template to create the job's `config.xml` file. We will do that by adding two new resources in `modules/online_store/manifests/ci.pp`:

```
class online_store::ci inherits online_store {
  package { ... }
  class { 'jenkins': ... }
  $plugins = ...
  jenkins::plugin { ... }
  file { '/var/lib/jenkins/hudson.tasks.Maven.xml': ... }
  $job_structure = ...
```

```
  $git_repository = ...
  $git_poll_interval = ...
  $maven_goal = ...
  $archive_artifacts = ...

  file { $job_structure:
    ensure  => 'directory',
    owner   => 'jenkins',
    group   => 'jenkins',
    require => Class['jenkins::package'],
  }

  file { "${job_structure[1]}/config.xml":
    mode    => 0644,
    owner   => 'jenkins',
    group   => 'jenkins',
    content => template('online_store/config.xml'),
    require => File[$job_structure],
    notify  => Service['jenkins'],
  }
}
```

The `File[$job_structure]` resource ensures that the job's directory structure is created while the `File["${job_structure[1]}/config.xml"]` resource uses the ERB template to create the job's configuration file and to notify that the Jenkins service should be restarted.

With these changes, we can test whether Puppet can really recreate the CI server from scratch. We do this using the `vagrant destroy ci` command to destroy the `ci` server, followed by a `vagrant up ci` command that it will provision it again:

```
$ vagrant destroy ci
    ci: Are you sure you want to destroy the 'ci' VM? [y/N] y
==> ci: Forcing shutdown of VM...
==> ci: Destroying VM and associated drives...
==> ci: Running cleanup tasks for 'puppet' provisioner...
$ vagrant up ci
```

```
Bringing machine 'ci' up with 'virtualbox' provider...
==> ci: Importing base box 'hashicorp/precise32'...
==> ci: Matching MAC address for NAT networking...
...
notice: Finished catalog run in 76.70 seconds
```

If all goes well, by the end of Puppet execution, you can access Jenkins' web interface at http://192.168.33.16:8080/ and, instead of seeing the welcome page, you will see that the online store job already exists and it will probably be running, as shown in figure 6.15.

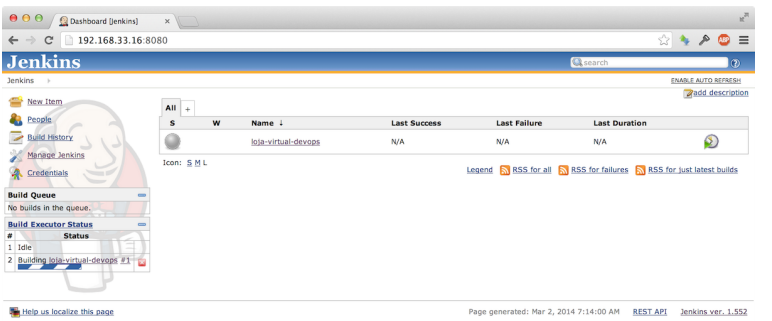

Fig. 6.15: Jenkins reprovisioned from scratch and running a new build

In this chapter we learned about some Agile engineering practices that enable us to develop quality software: version control systems, automated build processes, some of the main types of automated tests, and finally, how to provision a CI server to detect problems as soon as possible and ensure that the quality of the software is being evaluated on every commit.

Furthermore, we have found a better way to generate the `.war` artifact at the end of each build, outside of our production servers. However, the artifact is being archived only inside Jenkins. We need a way to join our build process with our deployment process to take the next step towards implementing

continuous delivery.

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 reposi-

tory 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
```

In this file, `Codename` is how Reprepro's command line tool refers to the repository by name. We use the ERB substitution syntax `<%= name %>` using the parameter `$name` from the `online_store::repo` class definition.

The `Architectures` setting lists which processor architectures are supported by packages in this repository. When a package includes binary files, the compilation process uses a different set of instructions depending on where the program will be executed, so it is important to define what architecture was used during compilation to guarantee that the package runs properly after its installation. In our case, the VMs are running Ubuntu in an `i386` processor.

The `SignWith` setting defines how Reprepro should sign packages published on that repository. We will use the `default` value for now and discuss more about package signing later on.

The next step is to create the repository's root directory and to define a new resource of type `File` to create the `distributions` configuration file using the newly created ERB template. We will do that in the `modules/online_store/manifests/repo.pp` file:

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

  $repo_structure = [
    "$basedir",
    "$basedir/conf",
  ]

  file { $repo_structure:
    ensure  => 'directory',
    owner   => 'jenkins',
    group   => 'jenkins',
    require => Class['jenkins'],
  }

  file { "$basedir/conf/distributions":
    owner  => 'jenkins',
```

```
    group  => 'jenkins',
    content => template('online_store/distributions.erb'),
    require => File["$basedir/conf"],
  }
}
```

The `$repo_structure` variable contains the list of directories that must be created. Initially, only the root and the `conf` directory are required. With that, we can define the `File["$basedir/conf/distributions"]` resource that will use the ERB template to create Reprepro's configuration file.

With the repository created, we need to expose its files so they can be accessed from other servers. We will use the **Apache** web server (http://httpd.apache.org/) so our repository can be accessed through HTTP. Installing and configuring Apache is quite simple if we use the Puppet Forge `apache` module, which is maintained by PuppetLabs. First of all, we will add the dependency to version `1.0.1` of the module in the `librarian/Puppetfile` file:

```
forge "http://forge.puppetlabs.com"

mod "puppetlabs/apt", "1.4.0"
mod "rtyler/jenkins", "1.0.0"
mod "puppetlabs/apache", "1.0.1"
```

With that, just modify the `modules/online_store/manifests/repo.pp` file to add the `apache` class and use the `apache::vhost` defined type to configure a virtual host:

```
class online_store::repo($basedir, $name) {
  package { 'reprepro': ... }
  $repo_structure = [...]
  file { $repo_structure: ... }
  file { "$basedir/conf/distributions": ... }

  class { 'apache': }

  apache::vhost { $name:
    port       => 80,
```

```
    docroot    => $basedir,
    servername => $ipaddress_eth1,
  }
}
```

In Apache, a **virtual host** is how you can run more than one website on the same web server. The `servername` and `port` parameters define the address and the port where an HTTP client can access the virtual host. The `docroot` parameter defines which directory will be exposed at that address. In our case, we will expose the `$basedir` directory on port 80 and will use the IP address of the server to access the repository.

The `$ipaddress_eth1` variable is not defined in this class and is not passed as a parameter. That variable is provided by Facter, a Puppet utility tool that harvests and exposes information about the server for use within a manifest file. To see all the available variables, you can run the `facter` command inside any of the virtual machines:

```
$ vagrant ssh ci
vagrant@ci$ facter
architecture => i386
...
ipaddress => 10.0.2.15
ipaddress_eth0 => 10.0.2.15
ipaddress_eth1 => 192.168.33.16
...
uptime_seconds => 11365
virtual => virtualbox
vagrant@ci$ logout
```

In this case, we have two network interfaces configured: `10.0.2.15` on the `eth0` interface corresponds to the default IP address assigned by Vagrant; `192.168.33.16` on the `eth1` interface corresponds to the IP address defined on our `Vagrantfile` so our servers can communicate through the network.

We can then reprovision the `ci` server running the `vagrant provision ci` command and, if all goes well, you can access the URL http://192.168.33.16/ in your browser and see the empty repository, as shown in figure 7.1.

Fig. 7.1: Empty package repository created

Creating and publishing a native package

With the repository ready, we need to change our build process to create a native package and then publish it. We will use **FPM** (https://github.com/jordansissel/fpm/) to do that, a Rubygem written by Jordan Sissel that simplifies the process of creating native packages for the different platforms – `.deb` in Debian/Ubuntu, `.rpm` on RedHat/CentOS, etc.

To install FPM we will add a new `Package` resource at the beginning of the `online_store::ci` class in the `modules/online_store/manifests/ci.pp` file:

```
class online_store::ci {
  include online_store

  package { ['git', 'maven2', 'openjdk-6-jdk', 'rubygems']:
    ensure => "installed",
  }

  package { 'fpm':
    ensure   => 'installed',
    provider => 'gem',
```

```
    require  => Package['rubygems'],
  }
  ...
}
```

Puppet's `Package` resource will use the system's default package manager, but you can override that. In this case, we are explicitly defining the `gem` provider in order for Puppet to installs a RubyGem and not a native package. We also need to install the native `rubygems` package, which is an FPM dependency.

Now we can change the online store's build configuration to execute a command at the end of every successful build. These are the necessary changes at the end of the `module/online_store/templates/config.xml` XML file:

```
<?xml version='1.0' encoding='UTF-8'?>
<maven2-moduleset plugin="maven-plugin@2.1">
  ...
  <prebuilders/>
  <postbuilders>
    <hudson.tasks.Shell>
      <command>
        fpm -s dir -t deb -C combined/target --prefix \
        /var/lib/tomcat7/webapps/ -d tomcat7 -n devopsnapratica
         \ -v $BUILD_NUMBER.master -a noarch devopsnapratica.war
         \ &amp;&amp; reprepro -V -b <%= repo_dir %> includedeb
         \ <%= repo_name %> *.deb
      </command>
    </hudson.tasks.Shell>
  </postbuilders>
  <runPostStepsIfResult>
    <name>SUCCESS</name>
    <ordinal>0</ordinal>
    <color>BLUE</color>
    <completeBuild>true</completeBuild>
  </runPostStepsIfResult>
</maven2-moduleset>
```

Let's break down the changes step by step, starting at the end and going

backwards: the `<runPostStepsIfResult>` element was changed to inform Jenkins that it must run a new command whenever a build of that job completes and the result is successful. The new `<hudson.tasks.Shell>` element – inside the `<postbuilders>` element – was created to configure a shell command that Jenkins should run.

The command itself can be broken into two parts: The first part creates the `.deb` package using FPM, and the second part publishes the package to the repository using `reprepro`. We will explain each option passed to the `fpm` command:

- `-s`: specifies that the source is a set of files and/or directories;
- `-t`: specifies that the desired format is a `.deb` package;
- `-C`: specifies that FPM needs to enter the `combined/target` directory to find the files that will be packaged;
- `--prefix`: the target directory where the files should be placed when the package is installed;
- `-d`: specifies a dependency between our package and the `tomcat7` package, since we require the webapps directory to exist before our package can be installed;
- `-n`: the name of our `.deb` package.
- `-v`: the package version. In this case, we are using an environment variable defined by Jenkins `$BUILD_NUMBER` that corresponds to the current build's number. This allows us to produce a new version of the package for every new build;
- `-a`: the package architecture. In our case, we have no platform-dependent compiled files, so we use the value `noarch`;
- `devopsnapratica.war`: the last argument is the name of the file we want to package.

After all of these options for the `fpm` command, we use the syntax `&amp;&amp;` which is the way to represent `&&` inside an XML file. The `&` character is special in an XML file, so we need to escape it, using the equivalent `&`. After that, we invoke the `reprepro` command with the following options:

- `-V`: specifies that the command should be more verbose. This will make it print more explanatory messages while it is executing;
- `-b`: the base directory where the repository is configured. In this case, we are using the `$repo_dir` variable value defined in the manifest for the `online_store::ci` class;
- `includedeb`: this is Reprepro's command to publish a new package to the repository. This command has two parameters: the repository name, represented by the variable `$repo_name` and the `.deb` file that will be published.

With this change to the job's configuration file, we need to reprovision the `ci` server again, using the `vagrant provision ci` command.

At the end of the Puppet execution, you will need to manually trigger a new build to test our changes. You can do this by clicking the *"Build Now"* link on page http://192.168.33.16:8080/job/loja-virtual-devops/.

To our surprise, the new build fails with the error shown in figure 7.2. One of the error messages says *"Error: gpgme created no signature!"*, which tells us that the publishing step failed when the repository tried to sign the package using the GPG tool.

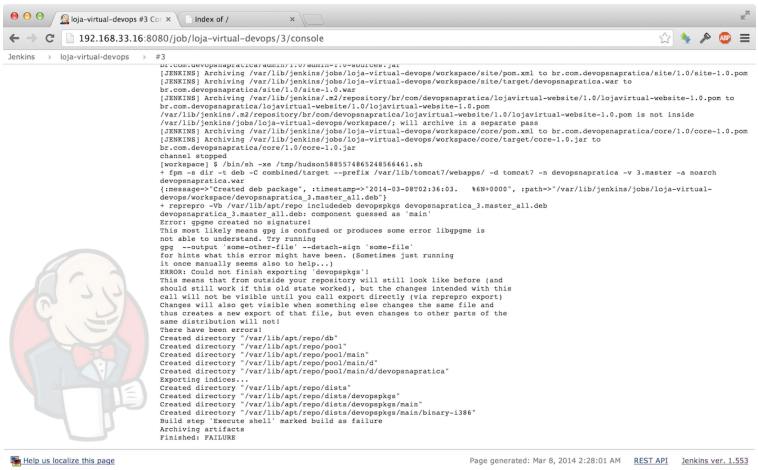

Fig. 7.2: Build failed trying to publish the package

GPG (http://www.gnupg.org/) – or *GnuPG* – is a tool for secure communication. Originally written to enable the safe exchange of e-mails, it relies on a key encryption scheme. Explaining this topic in depth is beyond the scope of the book, but it is enough to know that GPG allows you to generate key pairs, exchange and verify keys, encrypt and decrypt documents and authenticate documents using digital signatures.

Package repositories use GPG to digitally sign packages, and because of that we will need to configure a key pair. When a key pair is generated, we create a **public key** – which can be distributed for signature verification – and a **private key** – which must be protected and will be used by the repository to sign the packages when they are published.

The process to create keys uses the operating system's random number generator, but it requires a certain amount of entropy in the system to ensure that the encryption process remains robust. Entropy is generated when you use input devices while typing on the keyboard, moving the mouse, reading from the disk, etc. Since we are using a virtual machine, it is more difficult to generate entropy, so we will install a tool called **haveged** (http://www.issihosts.com/haveged/) that will help us in the key generation process:

```
$ vagrant ssh ci
vagrant@ci$ sudo apt-get install haveged
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following NEW packages will be installed:
  haveged
0 upgraded, 1 newly installed, 0 to remove and 157 not upgraded.
...
Setting up haveged (1.1-2) ...
```

As the repository is owned by the `jenkins` user, we need to generate GPG keys as that user. For that, we use the `su` command to switch to another user:

```
vagrant@ci$ sudo su - jenkins
jenkins@ci$ whoami
jenkins
```

Now we just need to run the `gpg --gen-key` command to generate the key pair. In the key generation process you will be asked a few questions. The first one is the type of key, for which we will use the `(1) RSA and RSA` option:

```
jenkins@ci$ gpg --gen-key
gpg (GnuPG) 1.4.11; Copyright (C) 2010 ...

Please select what kind of key you want:
   (1) RSA and RSA (default)
   (2) DSA and Elgamal
   (3) DSA (sign only)
   (4) RSA (sign only)
Your selection?
```

The next option is the key size, and we will use the default of 2048 bits:

```
RSA keys may be between 1024 and 4096 bits long.
What keysize do you want? (2048)
Requested keysize is 2048 bits
```

When asked about the key validity, we will use the option zero which defines that the key does not expire. Soon after this option, you must type `y` to confirm that you don't want the key to expire:

```
Please specify how long the key should be valid.
         0 = key does not expire
      <n>  = key expires in n days
      <n>w = key expires in n weeks
      <n>m = key expires in n months
      <n>y = key expires in n years
Key is valid for? (0)
Key does not expire at all
Is this correct? (y/N) y
```

Now you need to specify the name and email associated with this key pair. We will use the name *Online Store* and the email *admin@devopsinpractice.com*:

```
You need a user ID to identify your key; ...

Real name: Online Store
Email address: admin@devopsinpractice.com
Comment:
You selected this USER-ID:
    "Online Store <admin@devopsinpractice.com>"
```

At the end of all the questions, you need to choose option `O` to confirm all the information provided. GPG will then ask for a passphrase—a type of password to protect the key. In this case, we leave the passphrase blank, but this is not recommended in a production environment. As the signature process will be executed by the Jenkins job, we must confirm the blank passphrase:

```
Change (N)ame, (C)omment, (E)mail or (O)kay/(Q)uit? O
You need a Passphrase to protect your secret key.
...
pub   2048R/4F3FD614 2014-03-06
      Key fingerprint =
        C3E1 45E1 367D CC8B 5863  FECE 2F1A B596 ...
uid                  Online Store <admin@devopsinpractice.com>
sub   2048R/30CA4FA3 2014-03-06
```

When the command finishes you will see a summary of the newly created key pair. In particular, you need to remember the ID of the generated public key, which in this case is `4F3FD614`. The last thing we need to do is to export the public key to a file that will be available on the repository root:

```
jenkins@ci$ gpg --export --armor \
> admin@devopsinpractice.com > \
> /var/lib/apt/repo/devopspkgs.gpg
```

The `gpg --export` command exports the contents of the public key. The `--armor` option indicates which key pair we want to export, in this case using the email address `admin@devopsinpractice.com`. The output of this command is redirected to a new file – `/var/lib/apt/repo/devopspkgs.gpg` – that will be exposed along with the repository. Then, we can logout from the VM by executing the `logout` command twice: one to logout from the `jenkins` user and another to logout from the virtual machine:

```
jenkins@ci$ logout
vagrant@ci$ logout
$
```

Now we can run a new build by clicking the *"Build Now"* link again. This time the build should pass, as shown in figure 7.3.

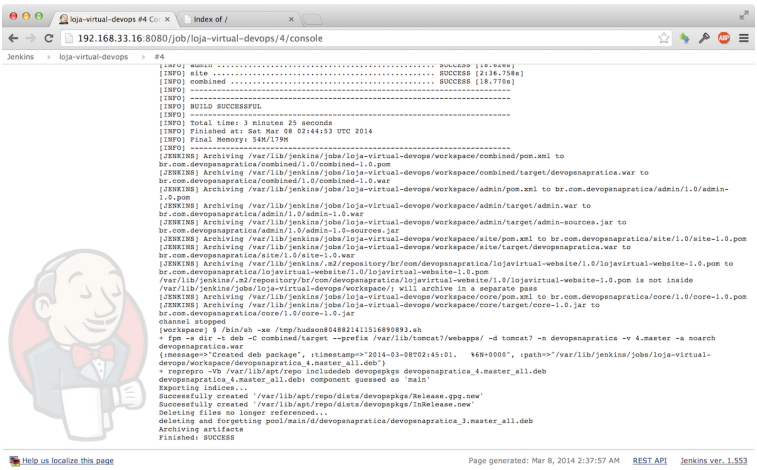

Fig. 7.3: Build passed and the package was published

You can verify that the package was published in the repository by accessing the URL http://192.168.33.16/ which should look something like figure 7.4.

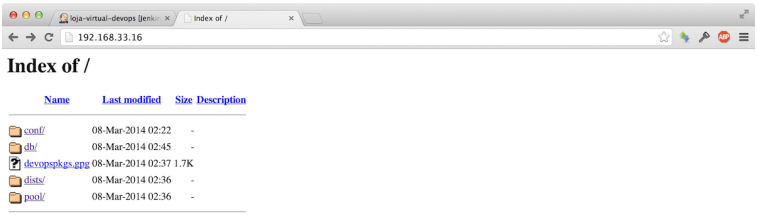

Fig. 7.4: Repository with the published package

Deploying with a native package

Now that we have a package repository with the online store's latest version, we can finally get rid of the `.war` file that is stored inside the Puppet module. For that, we will replace the `File['/var/lib/tomcat7/webapps/devopsnapratica.war']` resource with two new resources in the `online_store::web` class, at end of the `modules/online_store/manifests/web.pp` file:

```
class online_store::web {
  ...
  file { $online_store::params::keystore_file: ... }
  class { "tomcat::server": ... }

  apt::source { 'devopsinpractice':
    location    => 'http://192.168.33.16/',
    release     => 'devopspkgs',
    repos       => 'main',
    key         => '4F3FD614',
    key_source  => 'http://192.168.33.16/devopspkgs.gpg',
    include_src => false,
  }

  package { "devopsnapratica":
    ensure => "latest",
    notify => Service["tomcat7"],
  }
}
```

The `Apt::Source['devopsnapratica']` resource configures a new package repository that APT can use. The `location` parameter has the `ci` server IP address where the repository is exposed through HTTP. The `release` parameter is the desired repository name, while the `repos` parameter lists the desired components. The `key` and `key_source` parameters are signature verification settings, so we use the public key ID from the previous section and the URL to access the public key in the repository. Finally, the `include_src` parameter needs be set to `false` because our repository does not include packages with the source code of the distributed software.

With the repository configured, we create a new `package` resource to install the latest version of the `devopsnapratica` package and to notify the Tomcat service to be restarted after the package is installed. With that, we can reprovision the `web` server from scratch, destroying and rebuilding its VM:

```
$ vagrant destroy web
    web: Are you sure you want to destroy the 'web' VM? [y/N] y
==> web: Forcing shutdown of VM...
==> web: Destroying VM and associated drives...
==> web: Running cleanup tasks for 'puppet' provisioner...
$ vagrant up web
Bringing machine 'web' up with 'virtualbox' provider...
==> web: Importing base box 'hashicorp/precise32'...
==> web: Matching MAC address for NAT networking...
...
notice: Finished catalog run in 51.74 seconds
```

If all goes well, you should be able to access the online store in the URL http://192.168.33.12:8080/devopsnapratica/ and you can then delete the `devopsnapratica.war` file from the `modules/online_store/files` directory.

7.2 Continuous integration for the infrastructure code

We already have a way to publish and consume packages with new versions of the online store at the end of every build. However, all of the infrastructure code we wrote so far exists only in our local machine along with Vagrant's configuration file. We are not enjoying the benefits of continuous integration for our Puppet code.

In this section we will fix that by creating a new version control repository—a new build process—by adding automated tests and creating a new Jenkins job to execute this new build process, and publish a new package of the infrastructure code after each commit.

Creating a version control repository

The first decision is where to commit the infrastructure code we already have. We have two possibilities: put it in the application repository or create a new repository for the Puppet code. The advantage of putting everything in the same repository is that you simplify the setup process for a new member of the team because everything is in one place. However, you couple application changes with infrastructure changes: one commit in either of the two components will trigger a build process that generates both artifacts. As the lifecycle of application changes is relatively independent of the infrastructure, we will choose to use separate repositories.

You can initialize a new Git repository on the root directory where the Puppet modules and manifests are defined using the `git init` command, altering the `.gitignore` file to describe which files you do not want in the repository and making the first commit:

```
$ git init
Initialized empty Git repository in /tmp/devops-puppet/.git/
$ echo ".vagrant" >> .gitignore
$ echo "librarian/.tmp" >> .gitignore
$ echo "librarian/.librarian" >> .gitignore
$ echo "librarian/modules" >> .gitignore
$ echo "librarian/Puppetfile.lock" >> .gitignore
$ echo "*.tgz" >> .gitignore
$ git add -A
$ git commit -m"First commit"
[master (root-commit) 64686a5] First commit
 27 files changed, 664 insertions(+)
 create mode 100644 .gitignore
 ...
 create mode 100644 modules/tomcat/templates/server.xml
```

Then you can create a new GitHub repository manually, choosing a name – e.g. `loja-virtual-puppet`. After that, you can run the following command, substituting the `dtsato` username with your GitHub user and `loja-virtual-puppet` with the name you chose for your repository:

```
$ git remote add origin \
> git@github.com:dtsato/loja-virtual-puppet.git
```

```
$ git push -u origin master
Counting objects: 42, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (33/33), done.
Writing objects: 100% (42/42), 12.11 KiB | 0 bytes/s, done.
Total 42 (delta 1), reused 0 (delta 0)
To git@github.com:dtsato/loja-virtual-puppet.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```

The other option, which will save a few steps in the next sections, is to fork the existing repository from https://github.com/dtsato/loja-virtual-devops-puppet. Similar to what we did in chapter 6, you can fork this repository by clicking on the *"Fork"* button.

Regardless of your choice, we now have our Puppet code under version control and we can move forward to create an automated build.

A simple automated build

Puppet is written in Ruby and several of the tools used by the community are inspired or integrated with the Ruby ecosystem. Ruby has a very powerful, yet simple, build tool: **Rake** (http://rake.rubyforge.org/) . Inspired by Make, you can define **tasks** with prerequisites to execute arbitrary commands.

To understand how Rake works, let's create a `Rakefile` file where we can define our first tasks:

```
desc "Creates the puppet.tgz package"
task :package do
  sh "tar czvf puppet.tgz manifests modules librarian/modules"
end

desc "Removes the puppet.tgz package"
task :clean do
  sh "rm puppet.tgz"
end
```

The `package` task creates a package using the `tar` command, bundling all Puppet code that we have so far. The `clean` task deletes the generated

package using the `rm` command. To execute these tasks, you can just call the `rake` from the terminal:

```
$ rake -T
rake clean    # Creates the puppet.tgz package
rake package  # Removes the puppet.tgz package
$ rake package
tar czvf puppet.tgz manifests modules librarian/modules
a manifests
...
a librarian/modules/apache/files/httpd
$ rake clean
rm puppet.tgz
```

Notice that the `rake -T` command lists all the documented tasks. We know that the Librarian Puppet modules have already been downloaded, because they where included in the package. However, they would not be available if Librarian Puppet had not been executed previously. We can create another task to run the `librarian-puppet install` command and at the same time learn Rake's syntax for specifying prerequisites between tasks:

```
# encoding: utf-8
namespace :librarian do
  desc "Installs modules using Librarian Puppet"
  task :install do
    Dir.chdir('librarian') do
      sh "librarian-puppet install"
    end
  end
end

desc "Creates the puppet.tgz package"
task :package => 'librarian:install' do
  sh "tar czvf puppet.tgz manifests modules librarian/modules"
end
...
```

Rake also supports **namespaces** to group similar tasks with the same prefix. In this case, we created a namespace called `librarian` and an `install`

task inside it that simply changes into the `librarian` directory and runs the `librarian-puppet install` command. The `package` task definition also changed to declare its dependency on the new `librarian:install` task. When executing the `package` task again, we get an error:

```
$ rake package
librarian-puppet install
rake aborted!
Command failed with status (127): [librarian-puppet install...]
...
Tasks: TOP => package => librarian:install
(See full trace by running task with --trace)
```

This happens because Librarian Puppet is not installed on your system. We have just used it as a Vagrant plugin. In the Ruby world, the best way to manage this type of environment dependency is to use **Bundler** (http://bundler.io/) . With Bundler, you declare your dependencies in a `Gemfile` file and Bundler takes care of resolving the dependencies by downloading and installing Rubygems on the specified versions. Let's create the `Gemfile` file with the following content:

```
source 'https://rubygems.org'

gem 'rake', '10.1.1'
gem 'puppet-lint', '0.3.2'
gem 'rspec', '2.14.1'
gem 'rspec-puppet', '1.0.1'
gem 'puppet', '2.7.19'
gem 'librarian-puppet', '0.9.14'
```

Besides Rake and Librarian Puppet, we already added some other dependencies that will be needed in the next section. Also note that we are using specific versions of those Rubygems to make our local environment as close to the production environment as possible. For example, our VMs run Puppet `2.7.19`, so we use the same version in our build process.

To install the dependencies, you need to first install Bundler itself, in case you have never used it on your machine. With that done, you can run the `bundle install` command to install all necessary Rubygems:

```
$ gem install bundler
Successfully installed bundler-1.5.3
1 gem installed
$ bundle install
Fetching gem metadata from https://rubygems.org/...........
Fetching additional metadata from https://rubygems.org/..
Resolving dependencies...
Using rake (10.1.1)
...
Your bundle is complete!
Use `bundle show [gemname]` to see where a bundled gem is
installed.
```

Now the `package` task should run successfully. We just need to prefix the command with `bundle exec` to ensure that Rake will run using the gems on the versions we specified in our `Gemfile`:

```
$ bundle exec rake package
librarian-puppet install
tar czvf puppet.tgz manifests modules librarian/modules
a manifests
a manifests/ci.pp
...
```

We have an automated process to package our code, but we are not making any kind of testing. A simple and quick way to validate the Puppet code is to use the **Puppet Lint** tool (http://puppet-lint.com/) , which verifies that your code is following coding standards and generates warnings when issues are found. As we have already included it in our `Gemfile`, we can simply add a new task and dependency to the `Rakefile`:

```
# encoding: utf-8
require 'puppet-lint/tasks/puppet-lint'

PuppetLint.configuration.ignore_paths = ["librarian/**/*.pp"]

namespace :librarian do ... end

desc "Creates the puppet.tgz package"
```

```
task :package => ['librarian:install', :lint] do
  sh "tar czvf puppet.tgz manifests modules librarian/modules"
end

task :clean do ... end
```

Puppet Lint already defines a Rake task for us, so we just need to add a `require` of the `'puppet-lint/tasks/puppet-lint'` file. We also configured the task to ignore `.pp` files within the `librarian` directory because we don't want to validate the Puppet code of the external modules that we are using. Our build should focus on the modules that we are writing. Finally, we update the `package` task prerequisites to have two tasks using an `Array`.

When executing the `package` task, we will see several Puppet Lint warnings, represented as `WARNING` messages:

```
$ bundle exec rake package
librarian-puppet install
modules/online_store/manifests/ci.pp - \
WARNING: class not documented on line 1
modules/online_store/manifests/ci.pp - \
WARNING: double quoted string containing no variables on line 5
...
```

Testing Puppet code

Puppet Lint will find syntax and style problems in Puppet code, but it will not check if you typed the name of a wrong variable or whether a certain value is wrong. To test that, we can write automated tests for our modules using tools like **RSpec-Puppet** (http://rspec-puppet.com/) , an extension of the well known RSpec testing framework which gives us several extensions to simplify testing Puppet code.

The RSpec-Puppet Rubygem is also already installed, but we need to create a directory structure where we will put our test code within each module. First, we will focus on the `mysql` module:

```
$ cd modules/mysql
$ bundle exec rspec-puppet-init
 + spec/
```

```
 ...
 + Rakefile
$ rm Rakefile
$ cd ../..
```

Our first test will be for the `mysql::client` class, so let's create a new `client_spec.rb` file inside the `modules/mysql/spec/classes` directory with the following content:

```
require File.join(File.dirname(__FILE__), '..', 'spec_helper')

describe 'mysql::client' do
  it {
    should contain_package('mysql-client').
      with_ensure('installed')
  }
end
```

The `describe` block takes the name of the class, defined type or function of the module under test. Each `it` block represents a test case. In this case, the `mysql::client` class installs the `mysql-client` package so our test verifies if the package was installed. To run the test, we can use the `rspec` command:

```
$ bundle exec rspec modules/mysql/spec/classes/client_spec.rb
.

Finished in 0.34306 seconds
1 example, 0 failures
```

Success! We have our first passing test for our Puppet code. Like in chapter 6, our goal here is not to obtain a high test coverage, but to understand how to integrate testing into the continuous delivery process. For more details on how to write tests for Puppet code, you can explore the GitHub repository or read the RSpec-Puppet documentation on http://rspec-puppet.com/tutorial/.

What we want to do next is to run these tests as part of the build process. For that, we add new tasks to our `Rakefile`:

```
...
require 'rspec/core/rake_task'
TESTED_MODULES = %w(mysql)
namespace :spec do
  TESTED_MODULES.each do |module_name|
    desc "Runs tests for module: #{module_name}"
    RSpec::Core::RakeTask.new(module_name) do |t|
      t.pattern = "modules/#{module_name}/spec/**/*_spec.rb"
    end
  end
end

desc "Runs all tests"
task :spec => TESTED_MODULES.map {|m| "spec:#{m}" }

namespace :librarian do ... end

desc "Creates the puppet.tgz package"
task :package => ['librarian:install', :lint, :spec] do
  sh "tar czvf puppet.tgz manifests modules librarian/modules"
end

task :clean do ... end

task :default => [:lint, :spec]
```

Let's use the Rake task provided by RSpec to create a test task for each module. So far, we have only added tests for the `mysql` module. To run all tests, we created a `spec` task, which depends on each module's test tasks. We also added a new prerequisite for the `package` task and created a `default` task which will be executed when you run the `rake` command without specifying any task. For example:

```
$ bundle exec rake
modules/online_store/manifests/ci.pp - \
WARNING: class not documented on line 1
...
/Users/dtsato/.rvm/rubies/ruby-1.9.3-p385/bin/ruby \
-S rspec modules/mysql/spec/classes/client_spec.rb
```

```
.

Finished in 0.33303 seconds
1 example, 0 failures
```

We now have a build process and automated tests that we can use to implement continuous integration for our infrastructure code. We can then make a commit and push our code to the Git repository:

```
$ git add -A
$ git commit -m"Adding build and automated tests"
[master 3cf891c] Adding build and automated tests
 8 files changed, 100 insertions(+)
 create mode 100644 Gemfile
 ...
 create mode 100644 modules/mysql/spec/spec_helper.rb
$ git push
Counting objects: 21, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (12/12), done.
Writing objects: 100% (18/18), 2.34 KiB | 0 bytes/s, done.
Total 18 (delta 0), reused 0 (delta 0)
To git@github.com:dtsato/loja-virtual-puppet.git
   fe7a78b..3cf891c  master -> master
```

Configuring a Jenkins job for Puppet code

Now we can finally create a new Jenkins job for our Puppet code. Accessing Jenkins' home page at http://192.168.33.16:8080/, you can click on the *"New Item"* link. On the next page, we specify the name of the job as *"loja-virtual-puppet"* and the project type as *"Build a free-style software project"* as shown in figure 7.5.

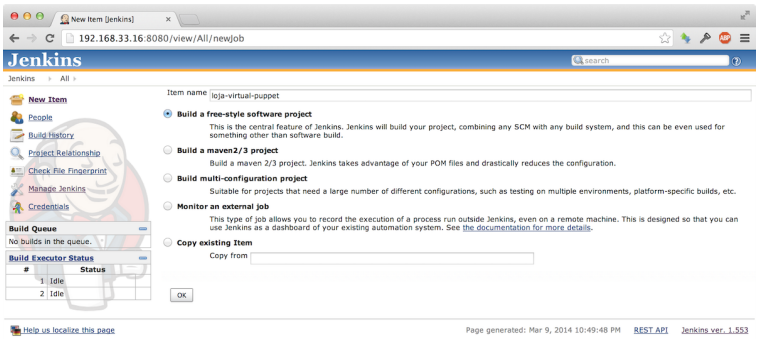

Fig. 7.5: New job for infrastructure code

By clicking the *"OK"* button, we arrive at the new job configuration screen. Let's describe step by step what we need to do in each section:

- ***"Source Code Management"* section**: we choose the *"Git"* option and fill out the *"Repository URL"* field with the repository's GitHub URL, created in the previous section, as shown in figure 7.6.

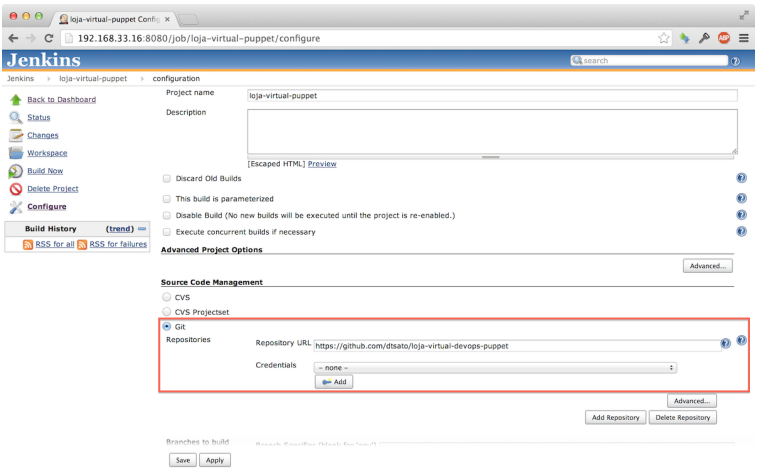

Fig. 7.6: Configuring the version control system

- ***"Build Triggers"* section**: we choose the *"Poll SCM"* option and fill out the *"Schedule"* text area with the value "* * * * *", as shown in figure 7.7

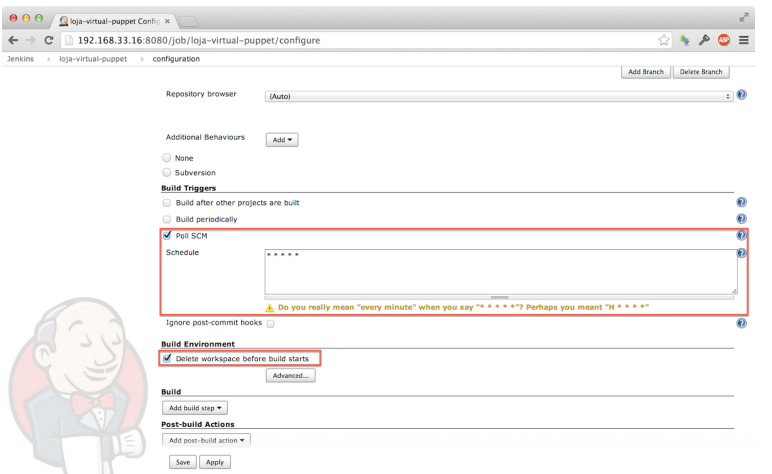

Fig. 7.7: Configuring job trigger and workspace cleanup

- ***"Build Environment"* section**: we choose the *"Delete workspace be-*

fore build starts" option to cleanup the workspace before starting a new build, as shown in figure 7.7.

- ***"Build"* section**: we choose the *"Execute shell"* option on the *"Add build step"* dropdown. The *"Command"* field must be filled out with the value `bundle install --path ~/.bundle && bundle exec rake package`, as shown in figure 7.8.

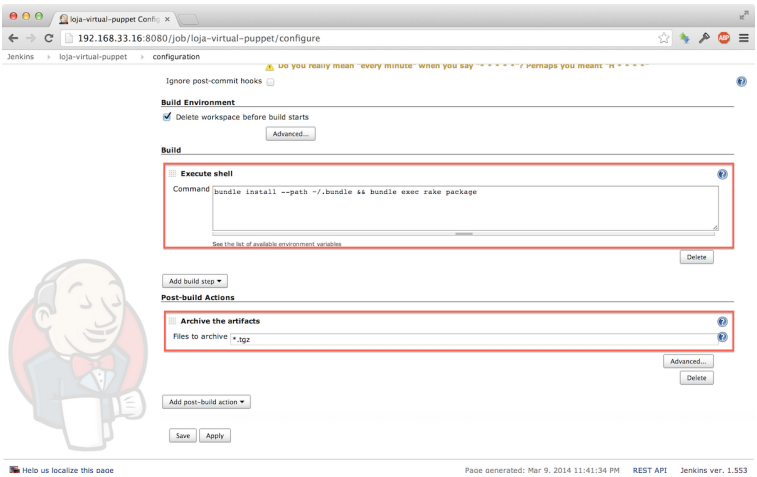

Fig. 7.8: Build command and post-build action to archive artifacts

- ***"Post-build Actions"* section**: by clicking the *"Add post-build action"* button we choose the *"Archive the artifacts"* option and fill out the *"Files to archive"* field with the value *"*.tgz"*, as shown in figure 7.8.

We can then click on the *"Save"* button and a new build should be scheduled and run in the next few minutes. However, the build fails with a message *"bundle: not found"*. We must install Bundler in order for Jenkins to run our build. We will do that by adding a new `package` resource at the beginning of the `modules/online_store/manifests/ci.pp` file, along with the FPM package:

```
class online_store::ci {
  ...
```

```
  package { ['fpm', 'bundler']:
    ensure   => 'installed',
    provider => 'gem',
    require  => Package['rubygems'],
  }
  ...
}
```

After reprovisioning the `ci` serve with the `vagrant provision ci` command, you can trigger a new build by clicking the *"Build Now"* link. This time it will finish successfully, as shown in figure 7.9.

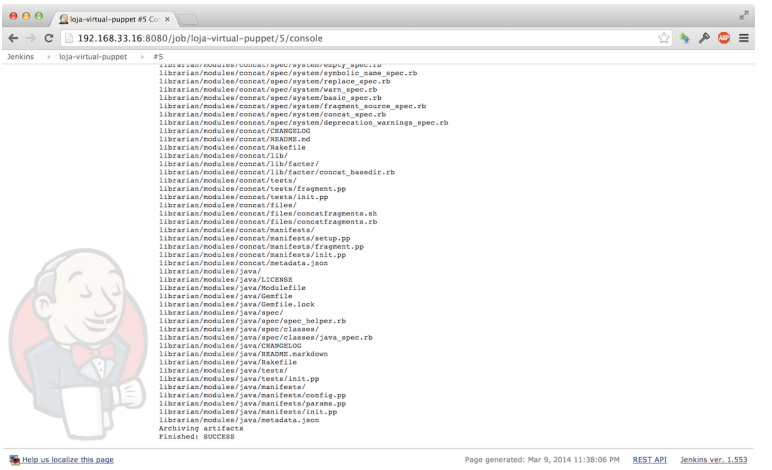

Fig. 7.9: Successfull build

Following the same steps from chapter 6, we can automate the process of creating that job with Puppet. We encourage you to try to that as an exercise to test your understanding of Puppet and the concepts presented so far. A solution is available on the GitHub repository you cloned at the beginning of this section. We will skip this explanation from the book to focus one of the most important continuous delivery concepts: the **deployment pipeline**.

7.3 Deployment pipeline

To finish this chapter, we will integrate our two builds and configure a one-click deployment process for production. Currently we have two independent CI processes, in which the final product of each build is a package that can go into production.

The path of a commit to production generally has to go through several steps— some of them are automated, while others require manual approvals. For example, it is common for companies to have a user acceptance phase where you can demonstrate the software running on a test environment and only after mutual consent the code can go into production. There are also several web companies that push any successful commit into production without any manual validation step.

It then becomes important to distinguish between two practices that are often confused: **continuous deployment** is the practice of putting into production every successful commit and usually allows several deploys to production per day; **continuous delivery** is the practice where any successful commit may go to production at any time, but the choice of when this happens is a business decision.

To illustrate the difference, think of two situations: in the first one you are developing a product that needs to be distributed and installed – for example, a mobile application – and to make a production deploy for every commit is not feasible or desirable. In another situation, you have a web application that is constantly used by multiple users and releasing fixes or new functionality quickly can give you a competitive advantage.

Regardless of which practice makes most sense to you, there is an essential pattern for implementing continuous delivery or continuous deployment: a **deployment pipeline**. On an abstract level, a deployment pipeline is an automated implementation of your application's delivery process, from a change in version control to its deployment in the production. It models the automated and manual steps of the delivery process and is a natural extension of the CI practice we have discussed so far.

The picture 7.10 shows an example of a deployment pipeline, where a change goes through a few stages: build and unit tests, acceptance tests, validation with the end user and deployment into production. The diagram also

shows how the pipeline provides quick feedback when problems are found. Each stage is there to veto the change before it gets to production. Only the commits that "survive" all pipeline stages are eligible to be deployed in production.

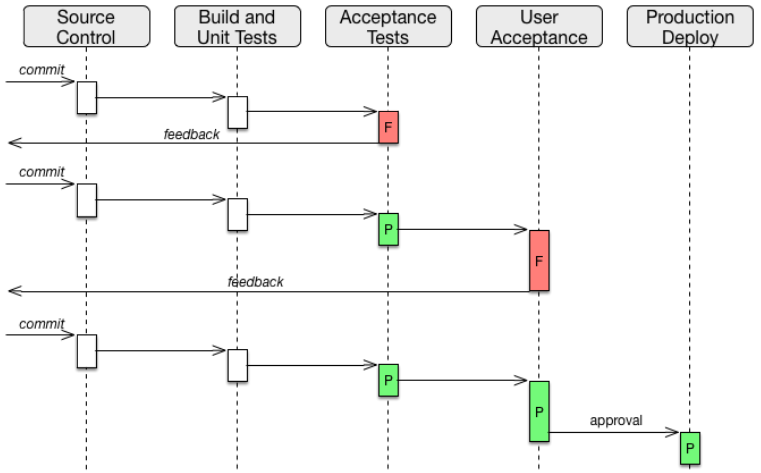

Fig. 7.10: Changes propagating through the deployment pipeline

Modeling your deployment pipeline requires understanding your delivery process and balancing the level of feedback you want to receive. Generally, the initial stages provide faster feedback, but may not be running exhaustive tests on a production-like environment. As the change propagates to later stages, your confidence increases, and environments become increasingly similar to production.

In our case, we will model an oversimplified deployment pipeline, since our goal is to show a practical implementation. For example, our functional tests currently run on the initial build of the application, along with the unit tests, but they are fast enough to run together. As the functional test coverage increases, usually the time to execute them also increases and you will need to consider breaking them into separate pipeline stages.

Figure 7.11 shows our deployment pipeline. We have two initial stages that run in parallel to build, test and package the application and the infrastructure code. Both stages are joined with a new stage to deploy to production,

which we will create next. For now, all transitions will be automatic so that our pipeline will implement a continuous deployment process.

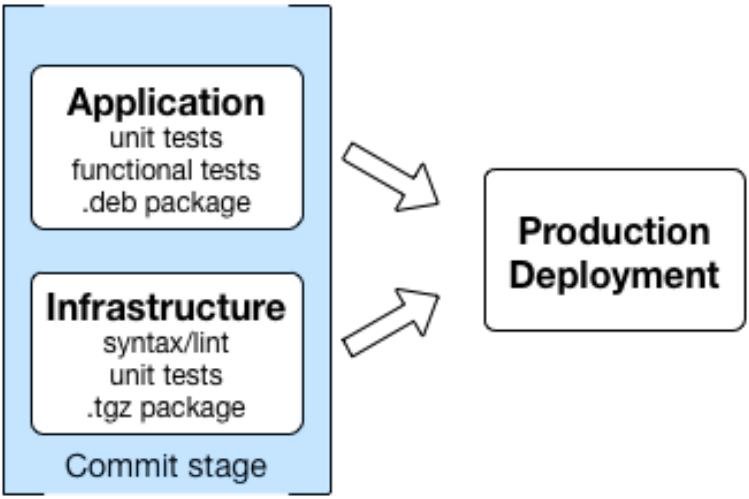

Fig. 7.11: Online store deployment pipeline

Creating a new production deployment stage

Before creating the new Jenkins job, we need to install two plugins that allow us to connect together our jobs: the *"Parameterized Trigger"* plugin (https://wiki.jenkins-ci.org/display/JENKINS/Parameterized+Trigger+Plugin) allows a build to be triggered when the another project's build finishes, as well as allows us to pass parameters between them; the *"Copy Artifact"* plugin (https://wiki.jenkins-ci.org/display/JENKINS/Copy+Artifact+Plugin) allows us to copy artifacts from one build to another.

We will add these plugins to the `online_store::ci` class, changing the `modules/online_store/manifests/ci.pp` file:

```
class online_store::ci {
  ...
  $plugins = [
    ...
    'ws-cleanup',
```

```
    'parameterized-trigger',
    'copyartifact'
  ]
  ...
}
```

We can then reprovision the `ci` server running the `vagrant provision ci` command and we will be ready to create the new Jenkins job. On the Jenkins home page, we will create a new job by clicking the *"New Item"* link and filling out the project name as *"Deploy to Production"* and choosing the project type *"Build a free-style software project"*.

Clicking *"OK"* will take us to the new job configuration page. This time we will not change the *"Source Code Management"* section, because we want our build to run when the other builds succeed. To do that, in the *"Build Triggers"* section we will choose the *"Build after other projects are built"* option and fill out the *"Project names"* field with the value *"loja-virtual-devops, loja-virtual-puppet"*.

In the *"Build Environment"* section we will again select the *"Delete workspace before build starts"* option. In the *"Build"* section, we will add two commands:

- **Copying Puppet artifact**: selecting the *"Copy artifacts from another project"* option, we fill out the *"Project name"* field with the value *"loja-virtual-puppet"* and select the *"Upstream build that triggered this job"* option. We will also select the *Use last successul build as fallback* option so we can trigger a production deploy manually without having to execute the upstream project's build. Finally, we fill out the *"Artifacts to copy"* field with the value *"puppet.tgz"*, as shown in figure 7.12.

- **Deploying to the web server**: we add another *"Execute shell"* command, filling out the *"Command"* field with a three-parts command, each separated by `&&`. The first part copies the artifact to the `web` server using `scp`. The second part executes a remote command on the `web` server to unpack the copied artifact. The third part will trigger a Puppet run using the `puppet apply` command and put our modules

in the `modulepath`. The full command, which can be seen in figure 7.12, is:

```
scp puppet.tgz vagrant@192.168.33.12: && \
ssh vagrant@192.168.33.12 'tar zxvf puppet.tgz' && \
ssh vagrant@192.168.33.12 'sudo /opt/vagrant_ruby/bin/puppet \
apply --modulepath=modules:librarian/modules manifests/web.pp'
```

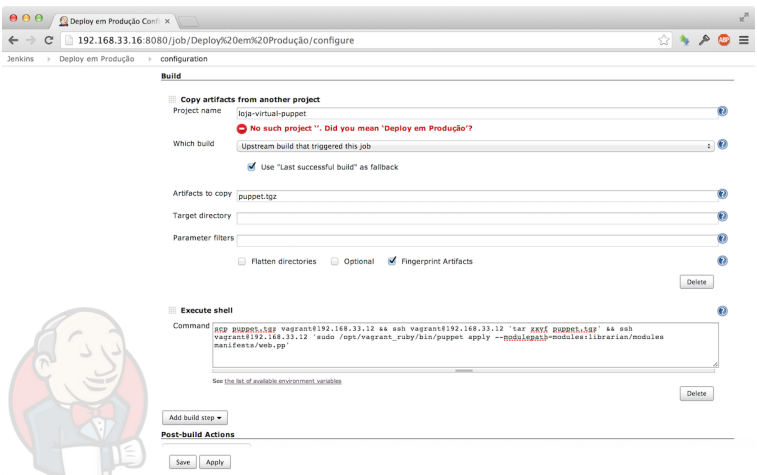

Fig. 7.12: Commands to deploy the online store

By clicking *"Save"* we have our new job to deploy to production. We will trigger a deploy by clicking the *"Build Now"* button of the new project. To our surprise, the deploy fails with an error *"Permission denied, please try again."*, as shown in figure 7.13.

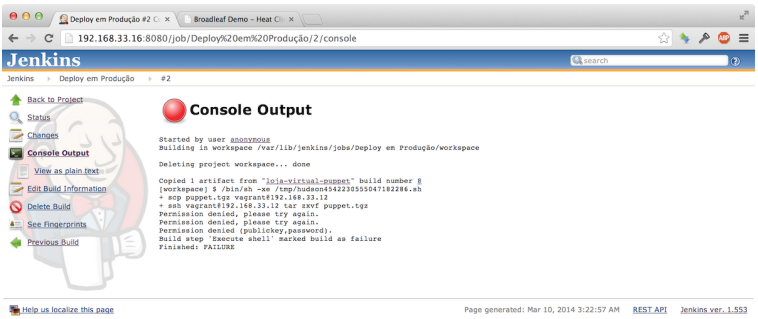

Fig. 7.13: Error trying to deploy

This happens because the `jenkins` user, that runs all the builds, does not have permission to copy or execute commands on the `web` server. To fix that, we need to login the VM and run some commands to exchange SSH keys between the servers:

```
$ vagrant ssh ci
vagrant@ci$ sudo su - jenkins
```

The `su` command changes the current user to `jenkins`. Then we can run the `ssh-keygen` command that will generate an SSH key pair and prompt us with a few questions:

```
jenkins@ci$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/var/lib/jenkins/.ssh/
                                                    id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /var/lib/jenkins/.ssh/
                                                    id_rsa.
Your public key has been saved in /var/lib/jenkins/.ssh/
                                                id_rsa.pub.
```

```
The key fingerprint is:
67:ae:7d:37:ec:de:1f:e7:c3:a8:15:50:49:02:9d:49
jenkins@precise32
The key's randomart image is:
+--[ RSA 2048]----+
|          .+E+o.  |
|            +o.   |
|             .    |
|              .   |
|        S o   .   |
|         +     .  |
|          .  oo..|
|         o  o.+=o|
|        . .o.+o.*|
+-----------------+
```

We use the default answer for all questions, including leaving the passphrase blank. With the SSH key pair created, we need to copy our public key to the `web` server. To do that, we will use the `ssh-copy-id` command with the `vagrant` user and the password is also `vagrant`:

```
jenkins@ci$ ssh-copy-id vagrant@192.168.33.12
vagrant@192.168.33.12's password:
Now try logging into the machine, with ...
```

With this, we finish the SSH configuration and we can logout from the VM, again executing the `logout` command twice:

```
jenkins@ci$ logout
vagrant@ci$ logout
$
```

We can now trigger a new deploy to production and this time everything will work out! Figure 7.14 shows the successful output of our deploy process.

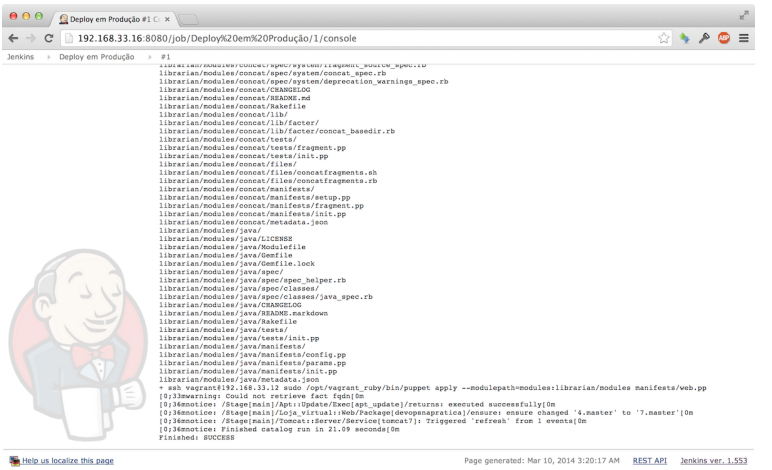

Fig. 7.14: Successful click-button deploy

7.4 Next steps

We now have an automated deployment pipeline that allows us to perform an automated deploy on every successful commit. As discussed earlier in the chapter, there are a few more things you can explore to improve your knowledge of the concepts we presented. For example, you can improve the Puppet code's test coverage.

Another interesting exercise is to automate the Jenkins settings that we performed manually in this chapter. The techniques shown in chapter 6 can help you with that task. Still on the topic of infrastructure as code, the `online_store::ci` class is starting to get complex, and after adding automation for configuring the new jobs, you might want to refactor it.

When we copy the `.tgz` file to the `web` server and uncompress it, we are leaving loose files around. It would be better to clean them up as soon as the deploy process ends to ensure that we leave nothing behind. If a module is removed in a subsequent build, we do not want to have problems in the deploy process for leaving the files around from a previous deploy.

Our decision to package the Puppet code as a `.tgz` file and copy it from a build to another using a Jenkins plugin is just one of the possible solutions.

There are other options to distribute Puppet code that were not discussed in the book, but you can always explore and learn about them. Puppet has a component called the Puppet Master that is a central server where hosts in your infrastructure register to keep themselves updated. But since that requires more infrastructure, we have decided to follow an approach without the Puppet Master and use a little more external orchestration.

Finally, our deploy process only updates the `web` server. We would like to make a full deploy of the entire infrastructure, or at least of the servers that directly affect the application's runtime environment. Another interesting exercise is to think about how to implement this deploy for the `db` server along with the `web` server.

Chapter 8

Advanced topics

We have already discussed several important concepts for adopting and implementing DevOps practices that support continuous delivery. Besides the collaboration between developers and operations, one of the most important cultural aspects of DevOps is continuous improvement. In the same way that retrospectives help Agile teams to adapt their development process, we need to use similar techniques to evaluate and evolve the software delivery process.

Continuous improvement is a way of applying the scientific method to your work. You start with an improvement hypothesis, based on your instinct or on observations and facts about your current process. You then propose an experiment to test your hypothesis. This implies collecting new data, facts, and observations that may support or refute your initial hypothesis. This learning cycle enables the implementation of small changes in your process to make it more and more efficient.

In this chapter we will cover some more advanced topics. Our goal is not

to create a comprehensive list of all techniques, tools, platforms, and methodologies that you should know. Instead, we want to cover a few important topics to give you enough knowledge to decide your next steps on your continuous improvement journey.

Evolving a manual deployment process into automation

In late 2012, I was working with a client who deployed to production once or twice a week. The client had a manual deployment process that had to be executed and closely monitored by one of the engineers, who stayed late at work to perform it. The testers also worked late hours to execute some basic manual tests right after each deploy.

One day, this engineer made a mistake and forgot to change a configuration file so that it used the correct database connection URL. This caused one of the indexing processes to use the wrong database. The process did not fail and the testers did not catch the mistake because the search feature was still working but returning incorrect results. Because of that, the problem was only detected the next day when a business representative was showing the product to a potential client.

As this failure had a high cost to the business, I facilitated a *post-mortem* meeting with the people involved and we did a **root cause analysis,** discussing the reasons of why the problem happened as well as potential measures to prevent it from happening again. One of the outcomes from that meeting was that I was to participate in the next deploy with the engineer in order to collect data about the manual processes that could be automated.

During the following week, I stayed late at work to follow the deploy process and I began writing down post-its for each manual step he executed. I also took note when he was not sure what to do next—the commands he executed to decide whether he could move forward or not. Late that night, we had an entire wall covered with post-its.

The next day, we had a meeting with the entire team and, using the post-its as input data, we generated several ideas on how to automate the manual tasks and some of the most problematic validations. We did not implement all the suggestions immediately, but we used this new knowledge to convince the business to allocate a certain amount of the team's capacity to work on

these improvements during upcoming iterations.

After a few months, the deployment process was almost fully automated and we found that several manual tasks could be executed and tested in other environments before running them in production. Despite the automated process, our client was still not comfortable performing deploys during the day, which is why we were still working late. However, by just pushing a button and waiting for the process to finish, we had far fewer surprises.

8.1 Deploying in the cloud

The cost of maintaining a datacenter with physical hardware is high for small and medium sized companies. Also, doing capacity planning in this scenario is difficult because you have to predict the growth rate even before you have your first user.

In recent years, companies have begun to invest in virtualization technologies to try to improve the utilization of the physical servers they already own. With virtual machines, you can run more than one server on the same physical machine, helping to control the purchasing and maintenance costs associated with hardware in the datacenter.

Meanwhile, internet companies like Amazon have invested a lot of money to create a robust infrastructure that can handle peak user access during the shopping season at the end of the year. The rest of the year, the servers were underutilized, leaving computing resources available. They then had the idea of renting these spare resources to potential customers, creating one of the most important technological innovations in the recent past: cloud computing.

The **cloud computing** model enables new ways of acquiring and paying for infrastructure, with five essential characteristics [9]:

- **On-demand self-service**: A consumer can provision computing infrastructure on demand in an automated way, without requiring human interaction with someone from the service provider.
- **Broad network access**: Cloud resources are available on the network and can be accessed by customers in any type of platform – mobile devices, tablets, laptops or other servers.

- **Resource pooling**: The provider's computing resources are part of a pool that can be used by multiple consumers. Generally, cloud resources abstract the actual location from consumers. In some cases, customers may specify the desired location, but it usually happens at a higher level of abstraction – as a certain country, state or datacenter.
- **Rapid elasticity**: Resources can be rapidly provisioned and released – in some cases automatically – so that the infrastructure can scale outward or inward according to demand. From the consumer's point of view, the cloud resources appear to be unlimited.
- **Measured service**: Cloud services automatically track and measure each resource's usage. Just like a water meter measures your monthly water usage, your provider will use resource usage data to charge you. The use of these resources can be monitored, controlled, and reported, providing transparency for both the provider and the consumer.

There are also three service models in cloud computing, with different abstraction levels:

- **IaaS** (Infrastructure as a Service): provides key infrastructure resources such as computing, storage and networking. The customer has enough control of these components, including choosing and accessing the operating system. Examples of IaaS providers are: AWS, OpenStack, DigitalOcean, Azure and Google Compute.
- **PaaS** (Platform as a Service): provides a managed platform where the customer can deploy their applications. The consumer is usually unable to control the infrastructure directly, only applications and possibly some configuration of the execution environment. Examples of PaaS offerings are: Heroku, Google App Engine, Elastic Beanstalk, OpenShift and Engine Yard.
- **SaaS** (Software as a Service): provides a product or service which runs on cloud infrastructure. The customer does not manage or control the infrastructure nor the application itself, only the product's configurations. Several online services can be considered SaaS: GMail, Dropbox, Evernote, Basecamp, SnapCI and TravisCI.

There are many public and private cloud compute providers. Amazon was the pioneer in this space with Amazon Web Services or AWS (http://aws.amazon.com/) , offering infrastructure as a service (IaaS) and a wide array of products and services. Despite being a paid service, it offers a free tier that can be used in the first year after you create a new account. Because of its flexibility and maturity, we will explore how to spin up our production environment on AWS.

Configuring the AWS environment

To create a new account on AWS is relatively simple and requires you to have a mobile phone at hand and a credit card. Amazon launches new products and offers very often, so this initial account creation process is always changing and getting simpler. For that reason, we will not describe it step by step, but we will assume that you have already configured and enabled your account.

One of the main AWS services is EC2, or *Elastic Compute Cloud*, which provides computing power on the cloud. With EC2, you decide how much computing power you need and provision the desired servers with an API call or the click of a button. It is a quick way of getting a new server in the cloud.

With the AWS account created, you can access the management console visiting https://console.aws.amazon.com/, and you will see the home page shown in figure 8.1. On this screen, you can have an idea of the amount of services offered by Amazon. EC2 is one of the links under the *"Compute & Networking"* section.

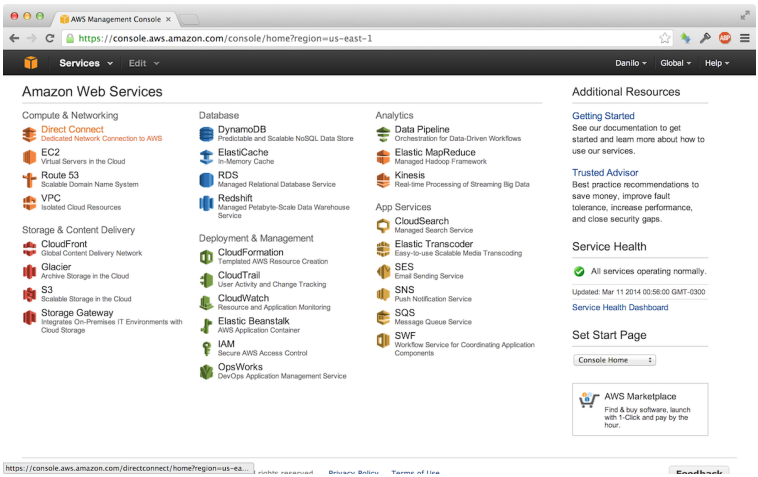

Fig. 8.1: AWS Console

Clicking on the EC2 link, you are taken to the EC2 control panel. The first thing we need to do is to create an SSH key pair to be able to access our servers once they are created. To do that, visit the *"Key Pairs"* link on the left side menu and click on the *"Create New Pair"* button as shown in the figure 8.2.

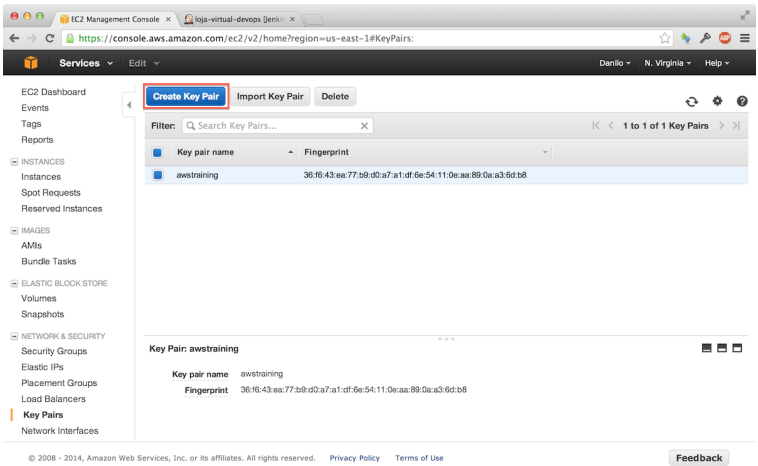

Fig. 8.2: Creating a new SSH key pair

Fill in the *"Key pair name"* field with the value *"devops"* and click *"Yes"*. The private key file `devops.pem` will be downloaded as shown in figure 8.3. You need to save it in a safe place because Amazon will not keep a copy of this file for you, as it would present a security problem. Take note of the path where you saved the primary key because we will use it in the next section. For now, we will assume that the file was saved on your `HOME` directory: `~/devops.pem`. With that we are able to access our servers via SSH.

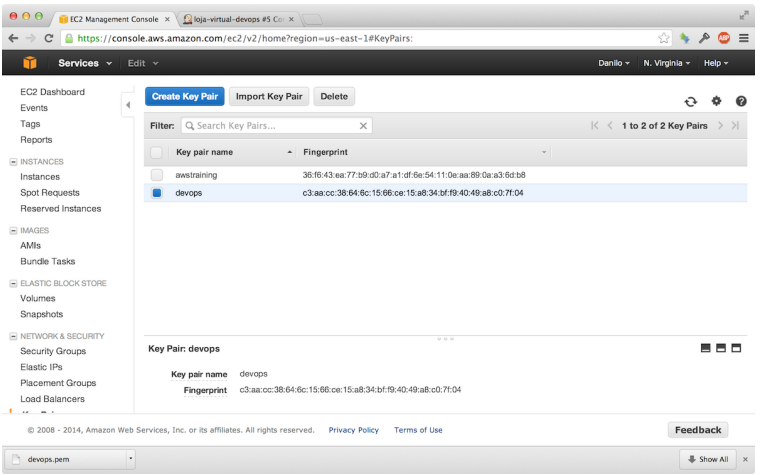

Fig. 8.3: Key pair created and downloading primary key

The next service we will use is VPC or *Virtual Private Cloud*. VPC enables us to create an isolated section of AWS where we can provision resources and have more control of the network configuration and topology. When launching an EC2 server in VPC, we can assign its internal IP address. With this service, we will be able to maintain the same network topology we used on our local environment in Vagrant and VirtualBox.

To create a private cloud on VPC, click on the *"VPC"* link on the *"Services"* dropdown menu at the top navigation bar. On the VPC control panel, click *"Get started creating a VPC"* as shown in figure 8.4.

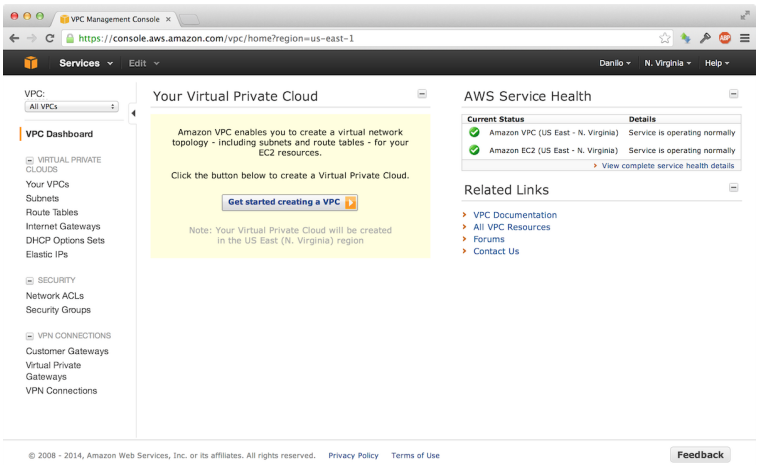

Fig. 8.4: VPC control panel

Choose the *"VPC with a Single Public Subnet Only"* option, as shown in figure 8.5. This will create a network topology with a single public network segment, with direct access to the internet.

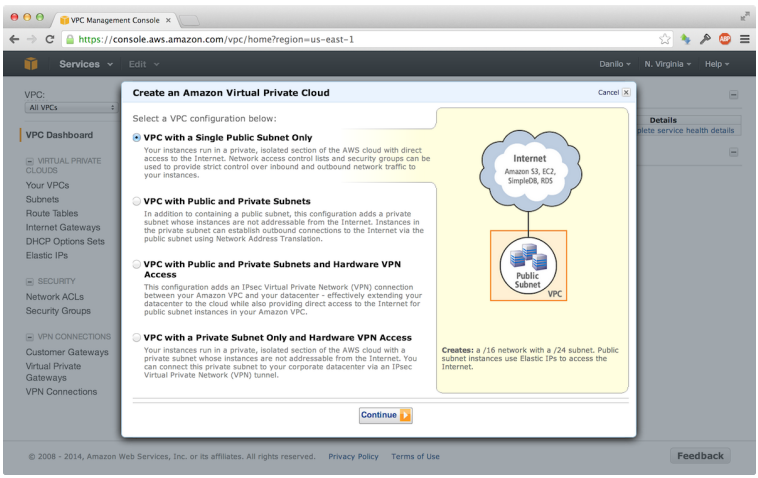

Fig. 8.5: Creating a new VPC

By clicking *"Continue"* you will be taken to the next setup screen where we will define our network segment. First of all, we will click on the *"Edit VPC IP CIDR Block"* link and fill out the *"IP CIDR block"* field with the value `192.168.0.0/16`. We will put the same value on the *"Public Subnet"* field after clicking the *"Edit Public Subnet"* link, as shown in figure 8.6.

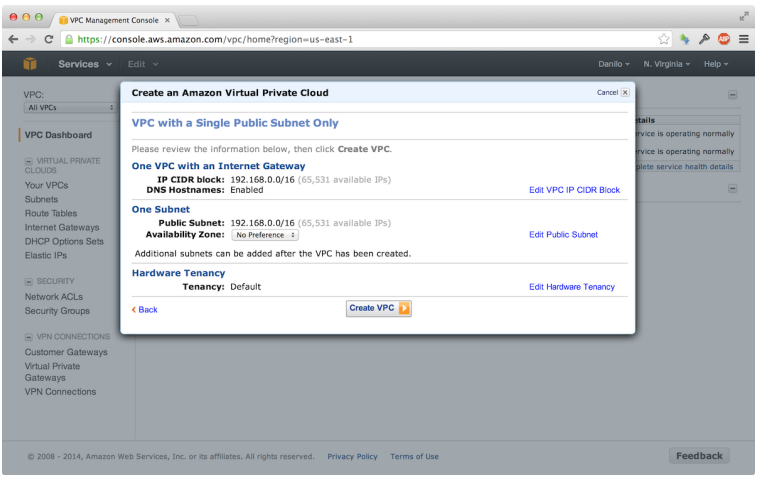

Fig. 8.6: VPC network settings

Finally, just click *'Create VPC'* to create our private cloud and we will see a success message as shown in figure 8.7.

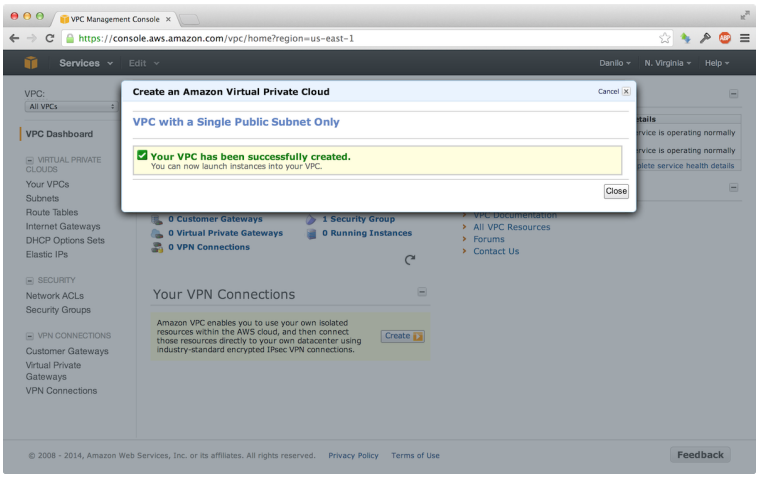

Fig. 8.7: VPC private cloud successfully created

We now need to write down the network subnet ID to use in the next section. You will find it by clicking on the *"Subnets"* link on the left menu and selecting the subnet we created. On this screen, you will see the network subnet details as shown in figure 8.8. Write down the ID, which will be different than the one highlighted in the figure, because AWS will assign a new unique ID every time a new resource is created.

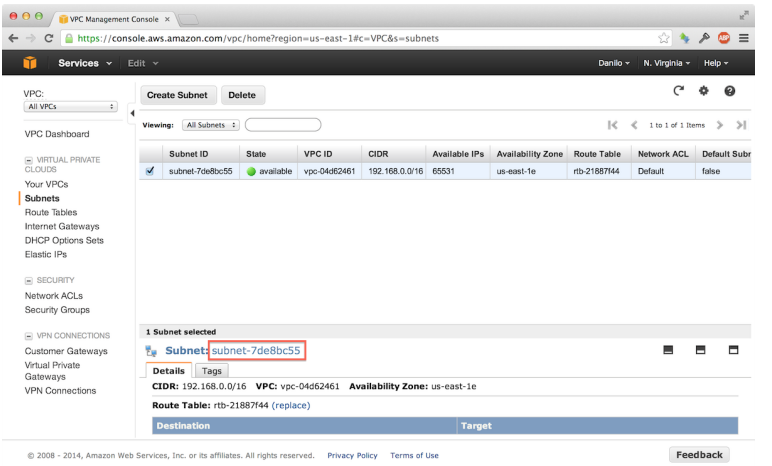

Fig. 8.8: Writing down the subnet ID

The last step we need to take in the AWS console is to set up a security group. A security group defines the access rules to EC2 instances. It is a kind of firewall that blocks or allows access in or out of the instance, depending on its configurations.

Let's click on the *"Security Groups"* link on the left side menu and select the *"default"* security group. On the details, click the *"Inbound"* tab to add 4 new rules that allow SSH, HTTP and HTTPS access on ports 22, 80/8080 and 443 respectively, as shown in figure 8.9. Click *"Apply Rule Changes"* for the changes to take effect.

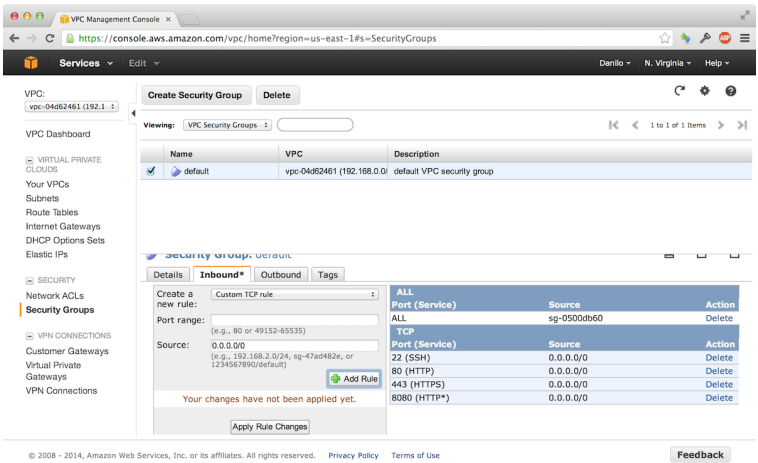

Fig. 8.9: VPC security group settings

Provisioning a production environment on AWS

To provision the production servers on AWS, we will use a Vagrant plugin called **vagrant-aws** (https://github.com/mitchellh/vagrant-aws) . Newer Vagrant versions support other providers, besides VirtualBox, and this plugin configures a provider that knows how to provision on AWS. Installing the plugin is simple:

```
$ vagrant plugin install vagrant-aws
Installing the 'vagrant-aws' plugin. This can take a few
                                                    minutes...
Installed the plugin 'vagrant-aws (0.4.1)'!
```

In addition, the plugin also requires us to import a different base box. As recommended by the plugin documentation, we will call this the box `dummy`. To add it, simply run the command:

```
$ vagrant box add dummy \
> https://github.com/mitchellh/vagrant-aws/raw/master/dummy.box
Downloading box from URL: https://github.com/mitchellh/...
Extracting box...e: 0/s, Estimated time remaining: --:--:--)
Successfully added box 'dummy' with provider 'aws'!
```

With the plugin installed and the new box available, we can change our Vagrant configuration file to add the AWS provider settings. We will do that on the `Vagrantfile` file:

```
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  config.vm.box = "hashicorp/precise32"
  ...
  config.vm.provider :aws do |aws, override|
    aws.access_key_id = "<ACCESS KEY ID>"
    aws.secret_access_key = "<SECRET ACCESS KEY>"
    aws.keypair_name = "devops"

    aws.ami = "ami-a18c8fc8"
    aws.user_data = File.read("bootstrap.sh")
    aws.subnet_id = "<SUBNET ID>"
    aws.elastic_ip = true

    override.vm.box = "dummy"
    override.ssh.username = "ubuntu"
    override.ssh.private_key_path = "<PATH TO PRIVATE KEY>"
  end

  config.vm.define :db do |db_config|
    ...
    db_config.vm.provider :aws do |aws|
      aws.private_ip_address = "192.168.33.10"
      aws.tags = { 'Name' => 'DB' }
    end
  end

  config.vm.define :web do |web_config|
    ...
    web_config.vm.provider :aws do |aws|
      aws.private_ip_address = "192.168.33.12"
      aws.tags = { 'Name' => 'Web' }
    end
  end
```

```
  config.vm.define :monitor do |monitor_config| ... end

  config.vm.define :ci do |build_config|
    ...
    build_config.vm.provider :aws do |aws|
      aws.private_ip_address = "192.168.33.16"
      aws.tags = { 'Name' => 'CI' }
    end
  end
end
```

You will need to make some changes using your own AWS configuration obtained in the previous section. The values for `subnet_id` and `private_key_path` were already captured on the previous section. To get the values for the `access_key_id` and `secret_access_key`, you can access the *"Security Credentials"* link on the user menu shown in figure 8.10.

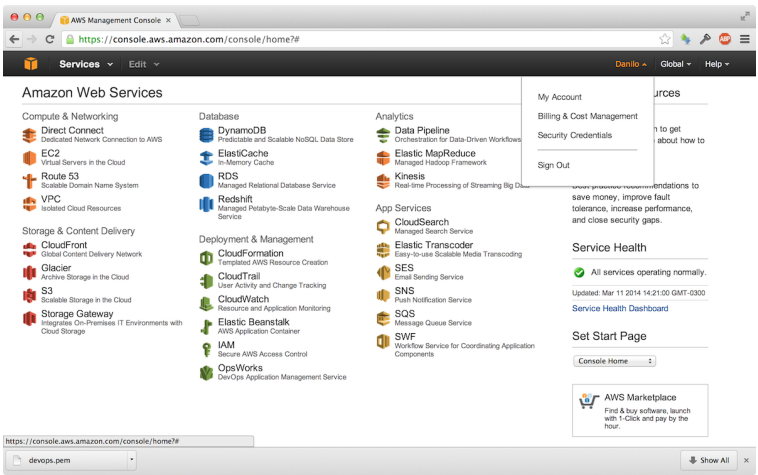

Fig. 8.10: Menu to access security credentials

On the next screen, after expanding the *"Access Keys"* section, you will see a *"Create New Access Key"* button as shown in figure 8.11.

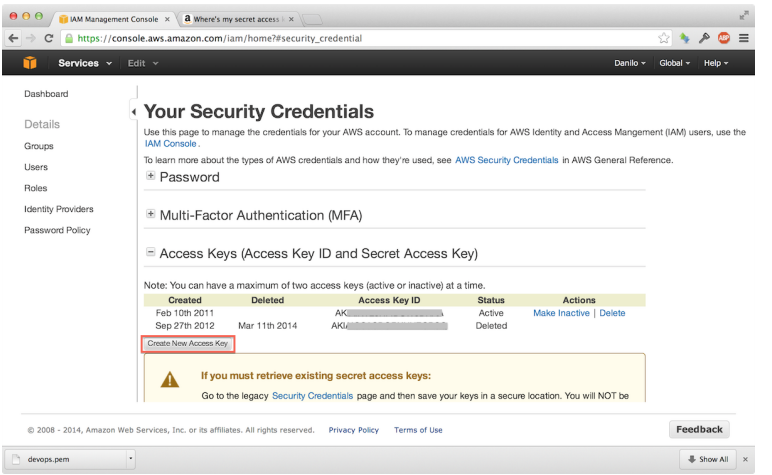

Fig. 8.11: Creating a new access key

After creating the access key successfully, you will see a new screen that will provide the values to fill out the `access_key_id` and `secret_access_key` fields, as shown in figure 8.12. These are your personal credentials— do not share them with anyone. If someone gets hold of your credentials, they will be able to launch new AWS resources and you will have to pay the bill!

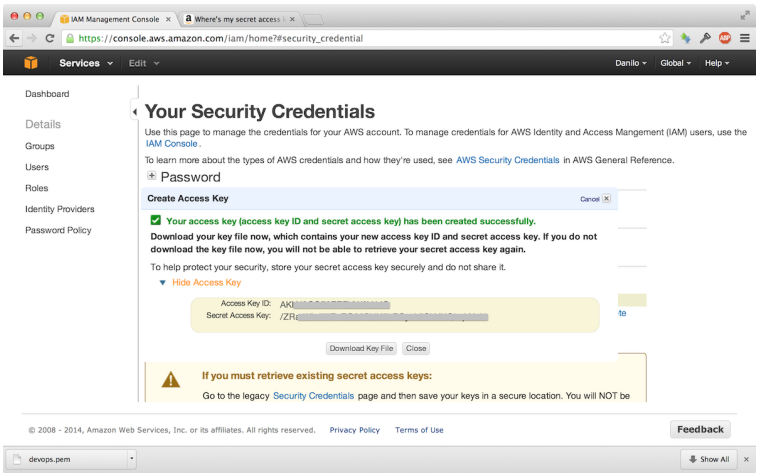

Fig. 8.12: Access key and secret key created successfully

On our `Vagrantfile`, we also defined other settings of the AWS provider. The `keypair_name` is the key pair name created in the previous section. The `ami` is the base image ID that AWS will use to launch a new server. In this case, the ID `"ami-a18c8fc8"` corresponds to an base image running a 32 bits Ubuntu 12.04 LTS, the same operating system version and architecture used in our VirtualBox VMs.

Finally, the `user_data` parameter points to a new file that we must create. When launching a new EC2 instance, you can pass a small script that will run as soon as the machine boots for the first time. In this case, we need a script to install Puppet, which is not pre-installed in the chosen AMI. The new `bootstrap.sh` file contents should be:

```
#!/bin/sh
set -e -x
export DEBIAN_FRONTEND=noninteractive

wget https://apt.puppetlabs.com/puppetlabs-release-precise.deb
dpkg -i puppetlabs-release-precise.deb

apt-get -y update
apt-get -y install puppet-common=2.7.19* facter=1.7.5*
```

This script is relatively simple. The `wget` and `dpkg -i` commands setup a new package repository with all the Puppet releases maintained by Puppet Labs. With that, we simply use the `apt-get` command to update the search index and install Puppet's base package on version 2.7.19, the same one we are using in our local VirtualBox VMs.

Okay, we can now provision our production servers on AWS with Vagrant using a new `--provider=aws` option, starting with the `db` server:

```
$ vagrant up db --provider=aws
Bringing machine 'db' up with 'aws' provider...
==> db: Installing Puppet modules with Librarian-Puppet...
==> db: Warning! The AWS provider doesn't support any of the
==> db: Vagrant high-level network configurations
==> db: (`config.vm.network`). They will be silently ignored.
==> db: Launching an instance with the following settings...
==> db:  -- Type: m1.small
==> db:  -- AMI: ami-a18c8fc8
...
notice: Finished catalog run in 39.74 seconds
```

If you see an error message saying that the server is already provisioned with a different provider, just run `vagrant destroy db ci web` to destroy your local VMs. In future Vagrant releases, you will be able to launch the same server using different providers at the same time. After destroying them, you can run the `vagrant up db --provider=aws` command again and it will work as expected.

We have our first server running in the cloud! Next up will be the `ci` server, because we need to publish the `.deb` package before launching the `web` server. But before doing that, let's make a small change in the `online_store::repo` class. In EC2, we will only have one `eth0` network interface, and our module assumes that the repository will be accessible in the `eth1` interface's IP address. We make this change at the end of the `modules/online_store/manifests/repo.pp` file:

```
class online_store::repo($basedir, $name) {
  package { 'reprepro': ... }
  $repo_structure = [...]
```

```
  file { $repo_structure: ... }
  file { "$basedir/conf/distributions": ... }
  class { 'apache': }

  if $ipaddress_eth1 {
    $servername = $ipaddress_eth1
  } else {
    $servername = $ipaddress_eth0
  }

  apache::vhost { $name:
    port       => 80,
    docroot    => $basedir,
    servername => $servername,
  }
}
```

Here we are using Puppet's syntax for conditional expressions. If the `$ipaddress_eth1` variable is defined, it will be used, otherwise we will fall back to `$ipaddress_eth0`. This allows our Puppet code to remain working locally with our VirtualBox VMs. We can now provision the `ci` server in the cloud:

```
$ vagrant up ci --provider=aws
Bringing machine 'ci' up with 'aws' provider...
==> ci: Installing Puppet modules with Librarian-Puppet...
==> ci: Warning! The AWS provider doesn't support any of the
==> ci: Vagrant high-level network configurations
==> ci: (`config.vm.network`). They will be silently ignored.
==> ci: Launching an instance with the following settings...
==> ci:  -- Type: m1.small
==> ci:  -- AMI: ami-a18c8fc8
...
notice: Finished catalog run in 423.40 seconds
```

After the server is provisioned, Jenkins will start a new online store build and will try to publish a new `.deb` package. If you remember what was necessary to setup the repository in chapter 7, we will need to login to the server

to generate the GPG key pair and export the public key. If you do not remember all the steps, please refer to the previous chapter, if you remember, follow the same instructions as shown below:

```
$ vagrant ssh ci
ubuntu@ip-192-168-33-16$ sudo apt-get install haveged
Reading package lists... Done
...
Setting up haveged (1.1-2) ...
ubuntu@ip-192-168-33-16$ sudo su - jenkins
jenkins@ip-192-168-33-16$ gpg --gen-key
gpg (GnuPG) 1.4.11; Copyright (C) 2010 ...
...
pub   2048R/3E969140 2014-03-11
      Key fingerprint =
        2C95 5FF8 9789 14F9 D6B3  ECD4 4725 6390 ...
uid                  Online Store <admin@devopsinpractice.com>
sub   2048R/67432481 2014-03-11
jenkins@ip-192-168-33-16$ gpg --export --armor \
> admin@devopsinpractice.com > \
> /var/lib/apt/repo/devopspkgs.gpg
jenkins@ip-192-168-33-16$ logout
ubuntu@ip-192-168-33-16$ logout
Connection to 54.84.207.166 closed.
$
```

Do not forget to write down the generated public key's ID, in this case `3E969140`, and replace the `key` parameter on the `Apt::Source[devopsinpractice]` resource in the `online_store::web` class. Now just wait for the build to complete and publish the `.deb` package. To follow the build progress, you need to find the Jenkins' server public host name on AWS. Visit the *"Instances"* link on the EC2 console's left side menu, select the "CI" server and copy the *"Public DNS"* field as shown in figure 8.13.

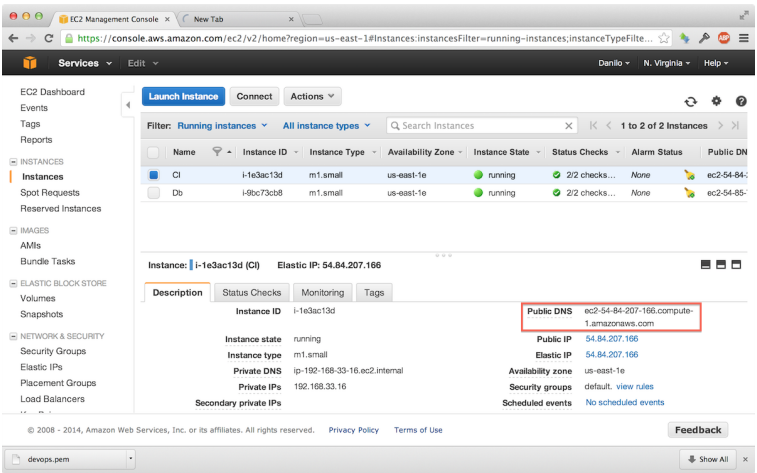

Fig. 8.13: CI server's public host name

With that address you can visit Jenkins on a new browser window, using port 8080. In this case, our URL is http://ec2-54-84-207-166.compute-1.amazonaws.com:8080/, and you will see Jenkins running in the cloud as shown in figure 8.14.

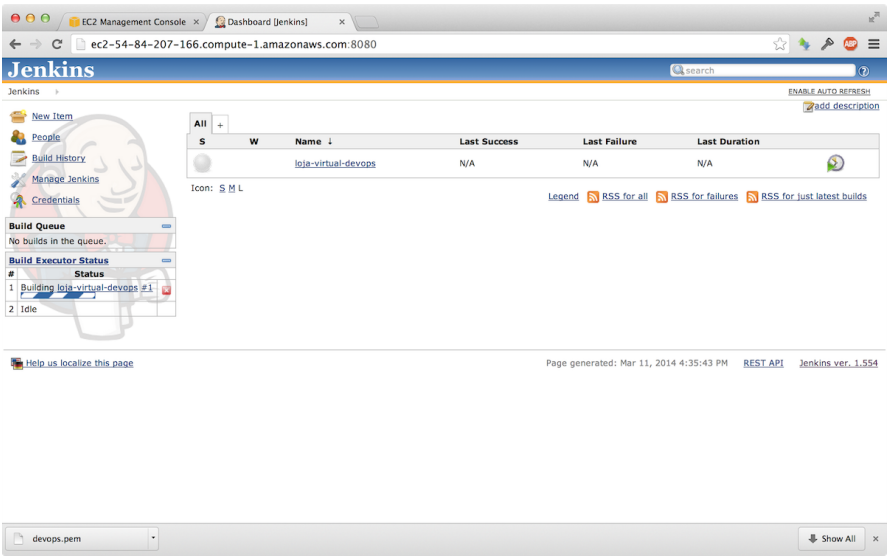

Fig. 8.14: Jenkins running in the cloud

Once our build finishes successfully, you can provision the `web` server:

```
$ vagrant up web --provider=aws
Bringing machine 'web' up with 'aws' provider...
==> web: Installing Puppet modules with Librarian-Puppet...
==> web: Warning! The AWS provider doesn't support any of the
==> web: Vagrant high-level network configurations
==> web: (`config.vm.network`). They will be silently ignored.
==> web: Launching an instance with the following settings...
==> web:  -- Type: m1.small
==> web:  -- AMI: ami-a18c8fc8
...
notice: Finished catalog run in 88.32 seconds
```

You can use the same procedure to find out the name of the `web` server's public hostname in AWS, as shown in picture 8.15.

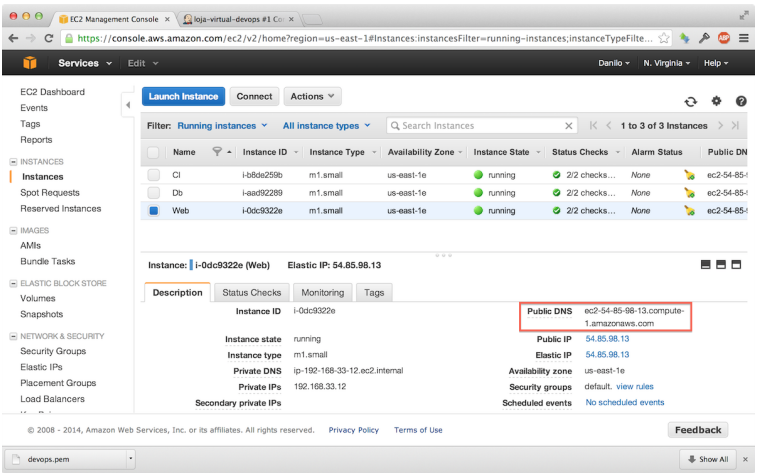

Fig. 8.15: Web server's public hostname

In our case, by accessing http://ec2-54-85-98-13.compute-1.amazonaws.com:8080/devopsnapratica/ we will see our online store running in the cloud, as shown in figure 8.16.

Success! We managed to migrate our application to the cloud without too much headache. The investment in automating the provisioning and deployment process has paid off.

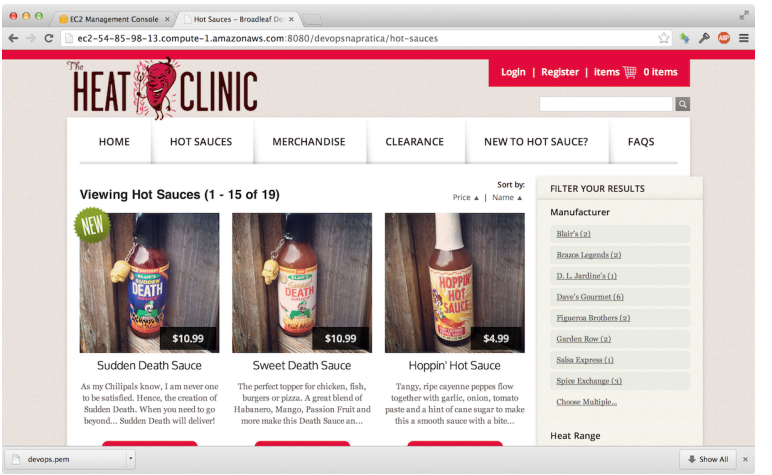

Fig. 8.16: Online store running in the cloud

As AWS resources are not free, we need to destroy our servers in order to save money. **Caution! If you skip this step, you will receive an unexpected bill at the end of the month!**

```
$ vagrant destroy ci web db
    db: Are you sure you want to destroy the 'db' VM? [y/N] y
==> db: Terminating the instance...
==> db: Running cleanup tasks for 'puppet' provisioner...
    web: Are you sure you want to destroy the 'web' VM? [y/N] y
==> web: Terminating the instance...
==> web: Running cleanup tasks for 'puppet' provisioner...
    ci: Are you sure you want to destroy the 'ci' VM? [y/N] y
==> ci: Terminating the instance...
==> ci: Running cleanup tasks for 'puppet' provisioner...
```

To ensure that the instances were actually terminated, access the *"Instances"* link on the EC2 control panel's left side menu and you should see something similar to figure 8.17.

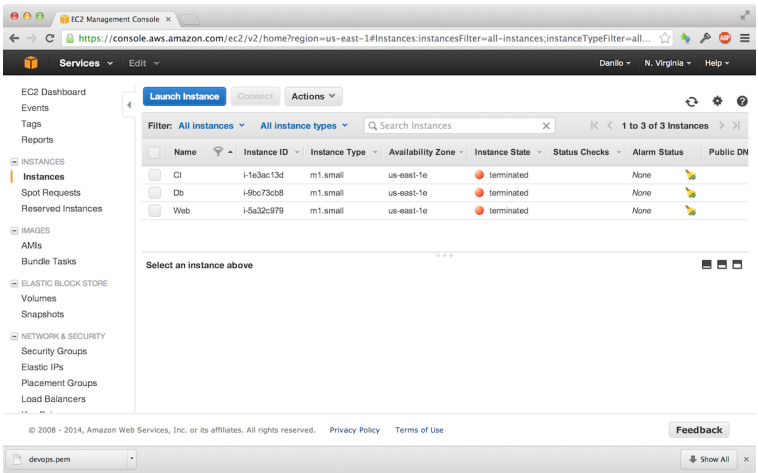

Fig. 8.17: EC2 instances terminated

8.2 DevOps beyond tools

It would be impossible to cover all the tools and techniques that are evolving in the DevOps space in this book. For all the topics we covered, there are several alternative tools being used by the community. Not only that, but there are also topics that were not covered and their respective tools.

The reason we used these tools was to demonstrate a practical example of implementing continuous delivery and DevOps. DevOps is more than your choice of tools. John Willis and Damon Edwards defined the acronym **CAMS** to define what DevOps is [19] and Jez Humble later introduced *Lean* to complete the acronym **CALMS** :

- **(C)ulture**: People and processes are more important. If culture is absent, any automation attempt is destined to fail.
- **(A)utomation**: Free humans to perform tasks that require creativity and intuition and let computers execute repetitive tasks, running them promptly and in a much more reliable way.
- **(L)ean (thinking)**: Many lean principles influenced the DevOps culture, such as continuous improvement, focus on quality, eliminating

waste, optimizing the whole and respect for people.

- **(M)easurement**: If you cannot measure, you will be unable to evaluate if something is getting better or worse. A successful DevOps adoption will measure as much as possible, for example: performance metrics, process metrics, business metrics etc.
- **(S)haring**: Collaboration and the sharing of ideas and acquired knowledge help to create the necessary culture for success with DevOps.

With these characteristics in mind, we will discuss some of the topics that were not covered in the book to give you pointers to continue exploring the DevOps space. Do not forget the last principle: Share what you learn and contribute with the community! Sharing stories is one of the best ways to increase knowledge.

8.3 Advanced monitoring systems

The monitoring system setup in chapter 3 focuses on detecting and reporting failures in the production environment. Nagios is a rather common tool in the industry, but it has its limitations: the graphical interface is outdated and difficult to use. Its configuration files are complicated and non-intuitive and in a more dynamic environment, where servers can come and go at any time, Nagios is not a very flexible tool.

There are other important aspects that should be considered in a more complete monitoring system. In particular, some areas are getting enough attention from the community and various tools and products are emerging to tackle them:

- **Log aggregation**: The more servers you have, the harder it is to investigate and debug problems. The traditional approach of logging into the server and running `grep` on the logs does not work because you need to be connecting to several servers to debug a single problem. For this reason, several tools are emerging to aggregate and index logs.

  A set of open source tools have gained popularity in this area: **Logstash** (http://logstash.net/) collects, filters, analyzes and stores log entries

from several different sources in **Elasticsearch** (http://elasticsearch.org/) , a search engine. Other options in this space are **Graylog2** (http://graylog2.org/) , **Splunk** (http://www.splunk.com/) , **Loggly** (https://www.loggly.com/) , **PaperTrail** (https://papertrailapp.com/) and **Logentries** (https://logentries.com/) .

- **Metrics**: I remember someone once told me that "you do not have a performance problem, you have a visibility problem." If you can see where and when the problem happens, it will be much easier for you to solve it. DevOps encourages you to measure as much as possible. You can capture metrics from the system, from the application, from the process, from the business or even from people. There are some tools evolving in this space that allow capturing and storing metrics. The **Metrics** library (http://metrics.codahale.com/) , written in Java by Coda Hale and with variants in other languages, provides a good API so developers can publish various metrics from the code. On the aggregation and storage side there are tools like **StatsD** (https://github.com/etsy/statsd/) , **CollectD** (http://collectd.org/) , **Carbon** (http://graphite.wikidot.com/carbon) , **Riemann** (http://riemann.io/) , **Ganglia** (http://ganglia.sourceforge.net/) , **TempoDB** (https://tempo-db.com/) , **OpenTSDB** (http://opentsdb.net/) and **Librato** (https://metrics.librato.com/) .

- **Visualizations**: The more data and metrics you collect, the higher the risk of your suffering from information overload. So it is important to invest in tools and data visualization techniques to synthesize large amounts of data in a more digestible format for humans. Some tools in that space are **Graphite** (http://graphite.wikidot.com/) **Cubism.js** (http://square.github.io/cubism/) , **Tasseo** (https://github.com/obfuscurity/tasseo) and **Graphene** (http://jondot.github.io/graphene/) .

- **Runtime information**: To understand where the bottlenecks are in your production application, you need to collect runtime information. With it you can find out how long your code is spending on external calls, rendering HTML pages, or which queries are taking

too much time in the database. Another area that has gained importance is capturing data from the user's point of view, for example: how long does your page take to load on the user's browser. Tools in this space are **NewRelic** (http://newrelic.com/) , **Google Analytics** (http://www.google.com/analytics/) and **Piwik** (http://piwik.org/) . Moreover, it is also important to capture errors using tool like **NewRelic** or **Airbrake** (https://airbrake.io/) .

- **Availability monitoring**: Another type of monitoring is knowing when your application is up or down. If you have users spread around the world, that kind of monitoring is even more important because you can be down in a few places but not others. Tools in this space are **Pingdom** (https://www.pingdom.com/) , **NewRelic**, **Monitor.us** (http://www.monitor.us/) and **Uptime Robot** (http://uptimerobot.com/) .

- **Analytics systems**: All these data can be consumed by an analytic system to try to find correlations, trends, or advanced insights about your system. There are not many tools in this space yet, but **NewVem** (http://www.newvem.com/) has some interesting analytic reports about your resource usage in the cloud to optimize your costs.

More holistic monitoring solutions try to attack all of those areas, allowing you to monitor at various levels and aggregatin information in only one place. With that, you can detect correlations between independent data points. **NewRelic** (http://newrelic.com/) is a paid SaaS product trying to go in that direction.

The more advanced and complete your monitoring system, the greater your trust in making more production deployments. Detecting operational problems becomes an extension – and in some cases even a replacement – of your testing process. IMVU, for example, achieves more than 50 deploys per day and created an immune system that monitors various business metrics during a deploy. When any regression problem is detected, this system reverts the last commit automatically and sends alerts to protect the system in production [6].

8.4 Complex deployment pipelines

In our example in chapter 7, we setup a very simplified deployment pipeline. In the real world, it is common that there are more stages and environments through which an application must pass before getting to production. In more complex cases, you will have various applications, components and services that integrate with each other. In these situations, you will need more levels of integration testing to ensure that components and services are working properly together.

Creating and managing more complex pipelines with Jenkins gets more complicated as the number of jobs grows. The pipeline is not a primary Jenkins concept, but can be modeled using triggers and job dependencies, as we did in the previous chapter. In that space, **Go.CD** (http://www.go.cd/) written by ThoughtWorks, was recently open sourced and always had the pipeline delivery as a core concept. With a distributed job execution model and a way to model different environments, Go is an attractive option to manage complex pipelines.

8.5 Managing database changes

One of the issues we have not addressed, but a very important one, is how to manage **database changes**. In our application, Hibernate creates the initial table schema and fills it up with reference data from the online store. In the real world, database schemas will evolve and you can not just erase and rebuild the database on every deploy.

To deal with database evolution, we usually try to decouple the application deployment process from the database deployment process. Because of its nature, some companies choose to update the database once a week while the application goes to production several times per day. Practices to evolve a database schema in a safe way are described in Pramod Sadalage's books: "Database Refactoring" [2] and "Recipes for continuous database integration" [14]. The "Continuous Delivery" book [11] also dedicates a chapter to the topic of managing data.

8.6 Deployment orchestration

Our deploy process is quite simple. It just reprovisions the `web` server automatically, whenever a new commit passes through the pipeline. The more components you have in your architecture, the greater will be the need to orchestrate the deployment order. In our case, for example, we would need to provision the database server before the web server. If your database has cluster replication, you will want to deploy in a certain order to prevent data loss and reduce the time during which the database is unavailable.

For these situations, it is recommended to create a script (or a set of scripts) responsible for deciding the order in which deployments should happen. Tools like **Capistrano** (http://capistranorb.com/) , **Fabric** (http://fabfile.org/) or even shell scripts may be used for this type of orchestration. Another alternative is to use **MCollective** (http://puppetlabs.com/mcollective) , a paid product offered by Puppet Labs.

One important technique to reduce deployment risk is known as **blue-green deployments**. If we call the current production environment "blue", the technique consist of introducing a parallel "green" environment with the new version of the software. Once everything is tested and ready to go live, you simply redirect all user traffic from the "blue" environment to the "green" environment. In a cloud computing environment, as soon as it is confirmed that the idle environment is no longer necessary, it is common to discard it, a practice known as **immutable servers.**

8.7 Managing environment configuration

It is common for your deployment pipeline to pass through various environments, for example: a testing environment, a UAT environment, a pre-production environment, and the production environment. An important principle is that you must **build your artifacts once and use them everywhere.** That means you must not put an environment-specific configuration in the main application artifact.

An example of this type of configuration is the database connection URL. In each environment you will have a different database instance running, so you need to configure the application to point to the right database. There are

a few different approaches to deal with this kind of problem:

- **Version control system**: With this approach you checkin a file with different environment configurations in the version control system. In Java, it is common to create `.properties` files for each environment. Your application then needs to know how to load the right file when it is running. The downside is that all configurations stay in the application, making it difficult to change them without performing a new deploy.
- **Configuration packages per environment**: This approach is similar to the previous one, but instead of packaging configuration files in the application artifact, you create packages for different environment-specific configurations. This allows you to change configuration without making a new application deploy.
- **DNS Configuration**: Instead of changing the URL for each environment, you can use the same URL in all environments – for example, `bd.app.com` – and use the DNS system to resolve the same hostname to the correct IP in that environment. This approach only works for URL-type configurations and it would not work for other types of configurations.
- **External provider settings**: In a few cases, you may want to control this type of configuration from a centralized location. A common concept in the operations world is a CMDB (Configuration Management DataBase), a repository of all IT assets of the organization and their respective settings. There are a few different ways to implement this practice: In Puppet, **hiera** (https://github.com/puppetlabs/hiera) stores configuration in a hierarchical manner, allowing you to override values in each environment and distribute them to each host in conjunction with Puppet master; in Chef you can create data bags to store these configurations or implement an LWRP (LightWeight Resource Provider) in order to get configuration from an external server. Another way to implement this practice is to leverage the version control system's *branching* and *merging* features, as described by Paul Hammant [10];

some companies end up implementing a customized solution to solve this problem.

- **Environment variables**: An approach used by Heroku that became popular when one of its engineers, Adam Wiggins, published the *"twelve-factor app"* article [18]. In this approach, every configuration that varies between environments is defined as an environment variable. This ensures that no configuration accidentally ends up in the code repository and is a solution that is language and operating system-agnostic.

There are advantages and disadvantages to each approach, so you will need to consider which ones make sense in your environment. For example, Heroku requires that configuration is provided in environment variables, so you have no other option. If you are controlling your entire infrastructure with Puppet, using hiera becomes more interesting.

8.8 Architecture evolution

At the end of the book, we finished with an environment comprising of four servers: the `web` and `db` servers are an essential part of the production environment that allows users to access the online store. The `monitor` and `ci` servers are part of our deployment and monitoring infrastructure. In the same way that we learn and improve our code design once it's in use, we will also learn and get feedback on our infrastructure once the application is in production.

As you invest in your monitoring system and obtain metrics (data and information on how your application behaves in production), you will have feedback to improve your infrastructure architecture. For example, the search functionality of our online store is provided by Solr and we are now running an embedded Solr instance running on the same JVM. As the quantity of indexed products increase, Solr will consume more memory and will compete for resources with Tomcat itself. In the worst case, a problem in Solr can affect the entire application.

An architecture evolution would be migrating Solr to a dedicated server. In that case, when a problem occurs, the application continues to run. You

only provide a degraded search functionality. Depending on the importance of this functionality, you can simply choose to disable it or use an alternative implementation using the database. All these are valid decisions and make your application more resilient from the user's point of view.

When you are planning your physical architecture and the expected capacity for your application, you must take into account several factors, usually associated with non-functional requirements:

- **Change frequency**: Keep together what changes at the same frequency. This principle applies both to software design as your system's architecture. If any component has more than one reason to change, it is an indication that it should be split. In our case, this was one of the motivations to separate the application's build stages, packages and repositories from the infrastructure.

- **Failure points**: It is important to know the failure points in your architecture. Even if you do not choose them purposefully, they will still exist. You just won't know where and when a failure can happen. In the embedded Solr example, we know that a failure in this component can bring down the entire application, so some architectural decisions must be taken with failure scenarios in mind.

- **Deployment and dependency units**: There is an architecture principle that says it is better to have "small pieces, loosely coupled" than a single monolith where all parts are highly coupled. When you think about your architecture, consider which components should be isolated from the rest and which ones can go to production independently from each other. In our case, this was another motivation to separate the `db` server from the `web` server.

- **Service-level agreements** (SLAs): an SLA is a metric that defines how a service should operate within preset limits. In some cases, providers associate financial penalties to SLAs. SLAs usually cover features such as performance and availability. When your application provides an SLA, you will need to make architectural decisions to ensure that you can achieve them. For example, if you provide a public API that needs

to be available 99.99% per year – that means it can only be unavailable at most 52 minutes and 33 seconds per year – you may need to add layers to limit the number of requests that a client can do, so that no customer can monopolize your resources or overwhelm your API. John Allspaw covers this subject in more detail in the book "The Art of Capacity Planning" [1].

*__Scalability__: Depending on your expectations about scalability, you will need to decide your architecture's extension points. There are two common approaches to dealing with this: **horizontal scalability** allows you to increase capacity by adding more servers in addition to what you already own; **vertical scalability** allows you to increase capacity by adding more resources – such as CPU, memory, disk, network, etc. – to the servers you already own. When you have separate components, you gain the possibility of considering these scalability requirements separately for each component. For example, you can choose to scale your web servers horizontally, while your database will scale vertically.

- **Costs**: The other side of the coin, when you separate more and more components, is cost. In cloud computing environments, you can optimize your costs as you learn more about how your application behaves in production. However, if you are planning to buy your own hardware, do not forget to consider the costs for each option before making any architectural decision.

- **Usage pattern**: If you know that your application has predictable usage spikes, or well-known seasonal characteristics, you can optimize your infrastructure to reduce costs. This is even more important in a cloud computing environment, where it is easy to increase or decrease the number of servers. In our case, the continuous integration server probably doesn't need to be available 24 hours a day, 7 days a week. If our developers work in the same office, it is quite possible that the server will not be used outside of business hours or on weekends. This is an advantage of migrating your infrastructure to the cloud, because you can just turn off the server and stop paying during the idle period.

- **Resource usage**: Finally, another characteristic that should be con-

sidered when you define your architecture is each component's usage pattern. Web servers generally use more CPU to serve multiple concurrent requests. On the other hand, database servers usually use more memory and disk for indexing and persistence. Knowing each of your system component's characteristics is important for choosing the best architecture.

There is no correct answer for every situation. As you gain more experience, you learn to identify which features are most important for your application. The principle of continuous improvement applies here as a way to evaluate whether your architectural choices are bringing the desired benefits.

8.9 Security

A very important subject in several aspects of your application and infrastructure is security. During the book, we chose weak passwords, and sometimes even left them blank to simplify our example. However, this is never the best option in a production environment. Another important warning is that you must not share secrets – passwords, passphrases, certificates or primary keys – between different environments. In some cases, it is even important that you do not share these secrets with all team members.

Never put secrets in plain text in the code repository. As a general rule, always try to encrypt sensitive information so that even when someone has access to the information, they are not able to obtain the password. In the same line of reasoning, it is important that you do not store personal user information in plain text. Do not let passwords appear in plain text in your logs. Do not share passwords by e-mail or write them on a post-it next to the server. Your system is only as secure as its weakest link and, as incredible as it may sound, the weakness is typically human.

Whenever we need to generate some sort of key, note that we did it manually so that we do not put the keys in the infrastructure configuration code. There are some alternatives to managing this type of information in popular infrastructure as code tools. In Puppet, you can use **hiera-gpg** (https://github.com/crayfishx/hiera-gpg) or **puppet-decrypt** (https://github.com/maxlinc/puppet-decrypt) . Chef has the concept of an encrypted data bag which can

reliably store secrets. In Java, the **Jasypt** library (http://www.jasypt.org/) provides a way to put encrypted settings in `properties` files so that they are not stored in plain text.

Another interesting principle that will help you find problems in your automated process is trying to avoid logging in to a production server to perform any type of operation. Blocking SSH access to the server removes an important attack vector, since no one (not even you) will be able to execute commands logging directly on the server.

It is clear that direct access is just one of the attack vectors. The best way to address security is to always use a layered scheme. Put protection on each level of your system: in the network components, in your application, in the software packages you choose, in your operating system, etc. It is also important that you keep informed about recent vulnerabilities and are prepared to patch components whenever a security issue is found and fixed.

Security is a complex issue and full of details, so it is important you seek more specialized resources. For web applications, a good reference for the most common vulnerabilities and ways to protect against them is the **OWASP** (https://www.owasp.org) . Every year, OWASP publishes a list of the top ten security risks to which your application can be exposed. They maintain a list of testing tools that can automatically verify some security issues that you can incorporate in your deployment pipeline.

8.10 Conclusion

One of the main objectives of this book was to introduce DevOps and Continuous Delivery techniques and concepts in practice. For this reason, we launched a non-trivial application in production as soon as the second chapter. We learned how to monitor and receive notifications when a problem is detected in production. We also saw how to treat infrastructure as a code, which allowed us to recreate the production environment quickly and reliably. We could even migrate our environment to the cloud without too much headache, reusing the code that we built throughout the chapters. We learned how to model our delivery process and implement an automated build and deployment process with different types of automated tests that give us the con-

fidence to go into production more often. Finally, with a deployment pipeline, we implemented a continuous deployment process in which every commit in the application or infrastructure code can go to production when it passes all tests.

This serves as a solid foundation that you can immediately apply when you go back to work tomorrow. As there are many more things to learn and the DevOps community continues to evolve and to create new tools, it would not be possible to tackle everything in just one book. Therefore, this chapter briefly discussed other important topics that you will encounter when embarking on a continuous delivery journey.

Our community is growing and the interest in the subject increases every day. Do not be alarmed if the tools presented here evolve or if new practices arise—this is how we create knowledge. The important thing is that you know where and how to look for resources, and that you share your learning with the community: write blog posts, submit a case study to a conference or in your user group, share your good and bad experiences. Help to improve our tools by contributing with code, bug fixes or improving their documentation, by sending questions and suggestions to the book's discussion group. Most importantly, stay connected!

Bibliography

[1] John Allspaw. *The Art of Capacity Planning: Scalling Web Resources*. O'Reilly Media, 2008.

[2] Scott Ambler and Pramod Sadalage. *Refactoring Databases: Evolutionary Database Design*. Addison-Wesley Professional, 2006.

[3] David J. Anderson. *Kanban: Successful Evolutionary Change for Your Technology Business*. Blue Hole Press, 2010.

[4] Kent Beck. *Test-Driven Development: By Example*. Addison-Wesley Professional, 2002.

[5] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 2004.

[6] James Birchler. Imvu's approach to integrating quality assurance with continuous deployment. 2010.

[7] Lisa Crispin and Janet Gregory. *Agile Testing: A Practical Guide for Testers and Agile Teams*. Addison-Wesley Professional, 2008.

[8] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[9] Timothy Grance and Peter M. Mell. The nist definition of cloud computing. Technical report, 2011.

[10] Paul Hammant. App config workflow using scm. 2013.

[11] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 2010.

[12] William E. Shotts Jr. *The Linux Command Line: A Complete Introduction*. No Starch Press, 2012.

[13] Steve Matyas Paul Duvall and Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley Professional, 2007.

[14] Pramod Sadalage. *Recipes for Continuous Database Integration: Evolutionary Database Development*. Addison-Wesley Professional, 2007.

[15] Danilo Sato. Uso eficaz de métricas em métodos Ágeis de desenvolvimento de software. Master's thesis, 2007.

[16] Ken Schwaber and Mike Beedle. *Agile Software Development with Scrum*. Prentice Hall PTR, 2001.

[17] ThoughtWorks. *ThoughtWorks Anthology: Essays on Software Technology and Innovation*. Pragmatic Bookshelf, 2008.

[18] Adam Wiggins. The twelve-factor app. 2012.

[19] John Willis. What devops means to me. 2010.