Subject: DEVOPS **B.Tech III year II sem.**

Class: CSM 3-2 BC

Unit-5 Notes

Testing the Code

Testing is very important for software quality, and it's a very large subject in itself.

We will concern ourselves with these topics in this chapter:

How to make manual testing easier and less error prone
Various types of testing, such as unit testing, and how to perform them in practice
Automated system integration testing

We already had a look at how to accumulate test data with Sonar and Jenkins in the previous chapter, and we will continue to delve deeper into this subject.

Manual testing

Even if test automation has larger potential benefits for DevOps than manual testing, manual testing will always be an important part of software development. If nothing else, we will need to perform our tests manually at least once in order to automate them.

Acceptance testing in particular is hard to replace, even though there have been attempts to do so. Software requirement specifications can be terse and hard to understand even for the people developing the features that implement those requirements. In these cases, quality assurance people with their eyes on the ball are invaluable and irreplaceable.

The things that make manual testing easier are the same things that make automated integration testing easier as well, so there is a synergy to achieve between the different testing strategies.

In order to have happy quality assurance personnel, you need to:

Manage test data, primarily the contents of backend databases, so that tests give the same results when you run them repeatedly
Be able to make rapid deployments of new code in order to verify bug fixes

Obvious as this may seem, it can be hard in practice. Maybe you have large production databases that can't just be copied to test environments. Maybe they contain end-user data that needs to be protected under law. In these cases, you need to de-identify the data and wash it of any personal details before deploying it to test environments.

Each organization is different, so it is not possible to give generally useful advice in this area other than the KISS rule: "Keep it simple, stupid."

Pros and cons with test automation

When you talk with people, most are enthusiastic about the prospect of test automation. Imagine all the benefits that await us with it:

Higher software quality
Higher confidence that the software releases we make will work as intended
Less of the monotonous tedium of laborious manual testing.

All very good and desirable things!

In practice, though, if you spend time with different organizations with complex multi-tiered products, you will notice people talking about test automation, but you will also notice a suspicious absence of test automation in practice. Why is that?

If you just compile programs and deploy them once they pass compilation, you will likely be in for a bad experience. Software testing is completely necessary for a program to work reliably in the real world. Manual testing is too slow to achieve Continuous Delivery. So, we need test automation to succeed with Continuous Delivery. Therefore, let's further investigate the problem areas surrounding test automation and see if we can figure out what to do to improve the situation:

Cheap tests have lower value.

One problem is that the type of test automation that is fairly cheap to produce, unit testing, typically has lower perceived value than other types of testing. Unit testing is still a good type of testing, but manual testing might be perceived as exposing more bugs in practice. It might then feel unnecessary to write unit tests.

It is difficult to create test cradles that are relevant to automated integration testing.

While it is not very difficult to write test cradles or test fixtures for unit tests, it tends to get harder as the test cradle becomes more production-like. This can be because of a lack of hardware resources, licensing, manpower, and so on.

The functionality of programs vary over time and tests must be adjusted accordingly, which takes time and effort.

This makes it seem as though test automation just makes it harder to write software without providing a perceived benefit.

This is especially true in organizations where developers don't have a close relationship with the people working with operations, that is, a non DevOps oriented organization. If someone else will have to deal with your crappy code that doesn't really work as intended, there is no real cost involved for the developers. This isn't a healthy relationship. This is the central problem DevOps aims to solve. The DevOps approach bears this repeating rule: help people with different roles work closer together. In organizations like Netflix, an Agile team is entirely responsible for the success, maintenance, and outages of their service.

It is difficult to write robust tests that work reliably in many different build scenarios.

A consequence of this is that developers tend to disable tests in their local builds so that they can work undisturbed with the feature they have been assigned. Since people don't work with the tests, changes that affect the test outcomes creep in, and eventually, the tests fail.

The build server will pick up the build error, but nobody remembers how the test works now, and it might take several days to fix the test error. While the test is broken, the build displays will show red, and eventually, people will stop caring about build issues. Someone else will fix the problem eventually.

It is just hard to write good automated tests, period.

It can indeed be hard to create good automated integration tests. It can also be rewarding, because you get to learn all the aspects of the system you are testing.

These are all difficult problems, especially since they mostly stem from people's perceptions and relationships.

There is no panacea, but I suggest adopting the following strategy:

Leverage people's enthusiasm regarding test automation
Don't set unrealistic goals
Work incrementally

Unit testing

Unit testing is the sort of testing that is normally close at heart for developers. The primary reason is that, by definition, unit testing tests well-defined parts of the system in isolation from other parts. Thus, they are comparatively easy to write and use.

Many build systems have built-in support for unit tests, which can be leveraged without undue difficulty.

With Maven, for example, there is a convention that describes how to write tests such that the build system can find them, execute them, and finally prepare a report of the outcome. Writing tests basically boils down to writing test methods, which are tagged with source code annotations to mark the methods as being tests. Since they are ordinary methods, they can do anything, but by convention, the tests should be written so that they don't require considerable effort to run. If the test code starts to require complicated setup and runtime dependencies, we are no longer dealing with unit tests.

Here, the difference between unit testing and functional testing can be a source of confusion. Often, the same underlying technologies and libraries are reused between unit and functional testing.
This is a good thing as reuse is good in general and lets you benefit from your expertise in one area as you work on another. Still, it can be confusing at times, and it pays to raise your eyes now and then to see that you are doing the right thing.

Selenium works by invoking a browser, pointing it to a web server running your application, and then remotely controlling the browser by integrating itself in the JavaScript and DOM layers.

When you develop the tests, you can use two basic methods:

Record user interactions in the browser and later save the resulting test code for reuse
Write the tests from scratch using Selenium's test API

Many developers prefer to write tests as code using the Selenium API at the outset, which can be combined with a test-driven development approach.

Regardless of how the tests are developed, they need to run in the integration build server.

This means that you need browsers installed somewhere in your test environment. This can be a bit problematic since build servers are usually headless, that is, they are servers that don't run user interfaces.

It's possible to wrap a browser in a simulated desktop environment on the build server.

A more advanced solution is using Selenium Grid. As the name implies, Selenium Grid provides a server that gives a number of browser instances that can be used by the tests. This makes it possible to run a number of tests in parallel as well as to provide a set of different browser configurations.

You can start out with the single browser solution and later migrate to the Selenium Grid solution when you need it.

There is also a convenient Docker container that implements Selenium Grid.

JavaScript testing

Since there usually are web UI implementations of nearly every product these days, the JavaScript testing frameworks deserve special mention:

Karma is a test runner for unit tests in the JavaScript language
Jasmine is a Cucumber-like behavior testing framework
Protractor is used for AngularJS

Protractor is a different testing framework, similar to Selenium in scope but optimized for AngularJS, a popular JavaScript user interface framework. While it would appear that new web development frameworks come and go everyday, it's interesting to note why a test framework like Protractor exists when Selenium is available and is general enough to test AngularJS applications too.

First of all, Protractor actually uses the Selenium web driver implementation under the hood.

You can write Protractor tests in JavaScript, but you can use JavaScript for writing test cases for Selenium as well if you don't like writing them in a language like Java
The main benefit turns out to be that Protractor has internalized knowledge about the Angular framework, which a general framework like Selenium can't really have.

AngularJS has a model/view setup that is particular to it. Other frameworks use other setups, since the model/view setup isn't something that is intrinsic to the JavaScript language—not yet, anyway.

Protractor knows about the peculiarities of Angular, so it's easier to locate controllers in the testing code with special constructs.

Testing backend integration points

Automated testing of backend functionality such as SOAP and REST endpoints is normally quite cost effective. Backend interfaces tend to be fairly stable, so the corresponding tests will also require less maintenance effort than GUI tests, for instance.

The tests can also be fairly easy to write with tools such as soapUI, which can be used to write and execute tests. These tests can also be run from the command line and with Maven, which is great for Continuous Integration on a build server.

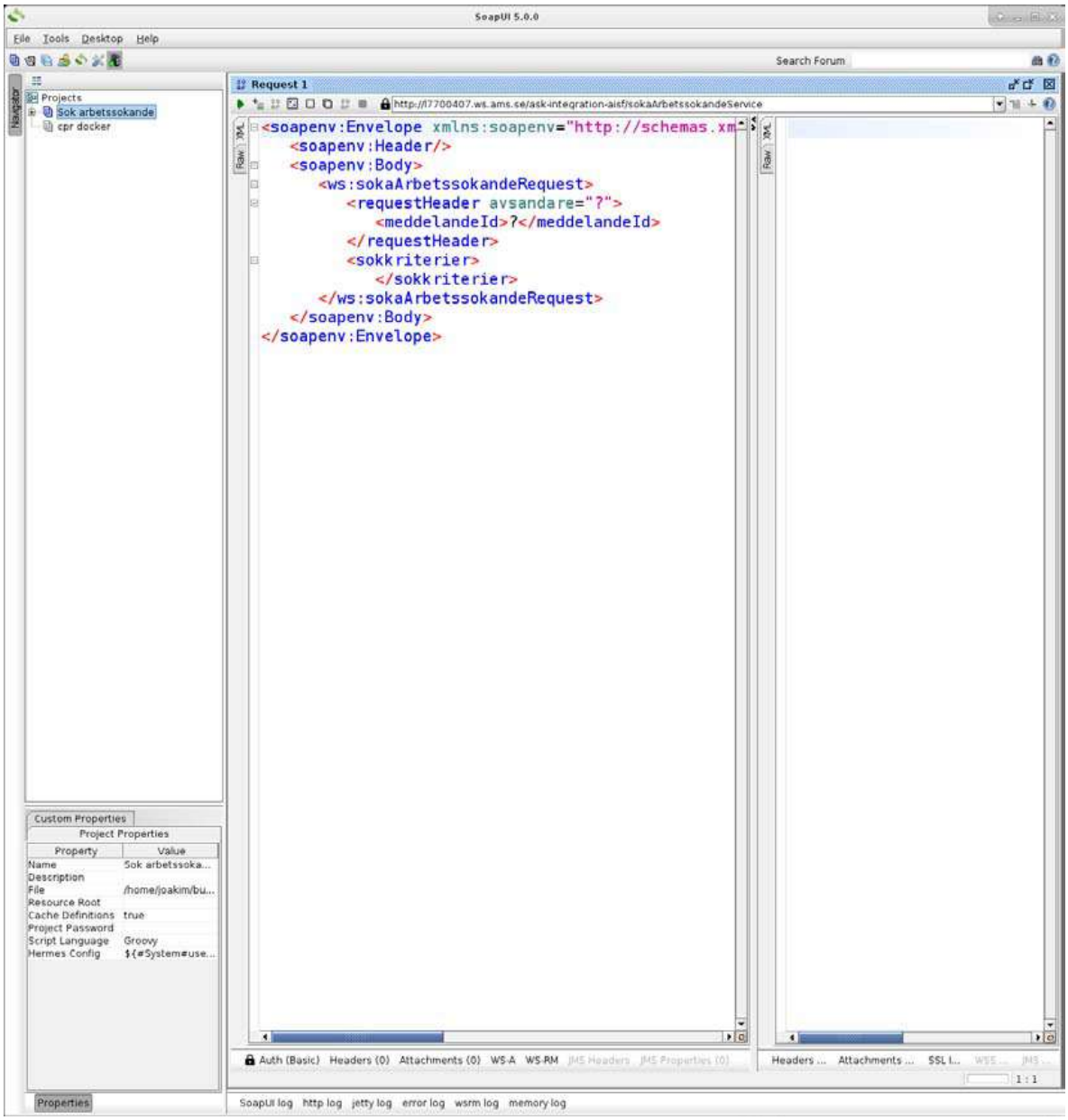
The soapUI is a good example of a tool that appeals to several different roles. Testers who build test cases get a fairly well-structured environment for writing tests and running them interactively. Tests can be built incrementally.

Developers can integrate test cases in their builds without necessarily using the GUI. There are Maven plugins and command-line runners.

The command line and Maven integration are useful for people maintaining the build server too.

Furthermore, the licensing is open source with some added features in a separate, proprietary version. The open source nature makes the builds more reliable. It is very stress-inducing when a build fails because a license has unexpectedly reached its end or a floating license has run out.

The soapUI tool has its share of flaws, but in general, it is flexible and works well. Here's what the user interface looks like:



The soapUI user interface is fairly straightforward. There is a tree view listing test cases on the left. It is possible to select single tests or entire test suites and run them. The results are presented in the area on the right.

It is also worth noting that the test cases are defined in XML. This makes it possible to manage them as code in the source code repository. This also makes it possible to edit them in a text editor on occasion, for instance, when we need to perform a global search and replace on an identifier that has changed names—just the way we like it in DevOps!

Test-driven development

Test-driven development (TDD) has an added focus on test automation. It was made popular by the Extreme programming movement of the nineties.

TDD is usually described as a sequence of events, as follows:

Implement the test: As the name implies, you start out by writing the test and write the code afterwards. One way to see it is that you implement the interface specifications of the code to be developed and then progress by writing the code. To be able to write the test, the developer must find all relevant requirement specifications, use cases, and user stories.

> The shift in focus from coding to understanding the requirements can be beneficial for implementing them correctly.

Verify that the new test fails: The newly added test should fail because there is nothing to implement the behavior properly yet, only the stubs and interfaces needed to write the test. Run the test and verify that it fails.

Write code that implements the tested feature: The code we write doesn't yet have to be particularly elegant or efficient. Initially, we just want to make the new test pass.

Verify that the new test passes together with the old tests: When the new test passes, we know that we have implemented the new feature correctly. Since the old tests also pass, we haven't broken existing functionality.

Refactor the code: The word "refactor" has mathematical roots. In programming, it means cleaning up the code and, among other things, making it easier to understand and maintain. We need to refactor since we cheated a bit earlier in the development.

TDD is a style of development that fits well with DevOps, but it's not necessarily the only one. The primary benefit is that you get good test suites that can be used in Continuous Integration tests.

REPL-driven development

While REPL-driven development isn't a widely recognized term, it is my favored style of development and has a particular bearing on testing. This style of development is very common when working with interpreted languages, such as Lisp, Python, Ruby, and JavaScript.
When you work with a Read Eval Print Loop (REPL), you write small functions that are independent and also not dependent on a global state.

The functions are tested even as you write them.

This style of development differs a bit from TDD. The focus is on writing small functions with no or very few side effects. This makes the code easy to comprehend rather than when writing test cases before functioning code is written, as in TDD.

You can combine this style of development with unit testing. Since you can use REPL-driven development to develop your tests as well, this combination is a very effective strategy.

Deploying the Code

Now that the code has been built and tested, we need to deploy it to our servers so that our customers can use the newly developed features!

There are many competing tools and options in this space, and the one that is right for you and your organization will depend on your needs.

We will explore Puppet, Ansible, Salt, PalletOps, and others, by showing how to deploy sample applications in different scenarios. Any one of these tools has a vast ecosystem of complementing services and tools, so it is no easy subject to get a grip on.

Throughout the book, we have encountered aspects of some of the different deployment systems that already exist. We had a look at RPMs and `.deb` files and how to build them with the `fpm` command. We had a look at various Java artifacts and how Maven uses the idea of a binary repository where you can deploy your versioned artifacts.

In this chapter, we will focus on installing binary packages and their configuration with a configuration management system.

Why are there so many deployment systems?

There is a bewildering abundance of options regarding the installation of packages and configuring them on actual servers, not to mention all the ways to deploy client-side code.

Let's first examine the basics of the problem we are trying to solve.
components. We don't need to make the scenario overly complex in order to start reasoning about the challenges that exist in this space.

In our scenario, we have:

A web server
An application server
A database server

If we only have a single physical server and these few components to worry about that get released once a year or so, we can install the software manually and be done with the task. It will be the most cost-effective way of dealing with the situation, even though manual work is boring and error prone.

It's not reasonable to expect a conformity to this simplified release cycle in reality though. It is more likely that a large organization has hundreds of servers and applications and that they are all deployed differently, with different requirements.

Managing all the complexity that the real world displays is hard, so it starts to make sense that there are a lot of different solutions that do basically the same thing in different ways.

Whatever the fundamental unit that executes our code is, be it a physical server, a virtual machine, some form of container technology, or a combination of these, we have several challenges to deal with. We will look at them now.

Virtualization stacks

Organizations that have their own internal server farms tend to use virtualization a lot in order to encapsulate the different components of their applications.

There are many different solutions depending on your requirements.

Virtualization solutions provide virtual machines that have virtual hardware, such as network cards and CPUs. Virtualization and container techniques are sometimes confused because they share some similarities.

You can use virtualization techniques to simulate entirely different hardware than the one you have physically. This is commonly referred to as emulation. If you want to emulate mobile phone hardware on your developer machine so that you can test your mobile application, you use virtualization in order to emulate a device. The closer the underlying hardware is to the target platform, the greater the efficiency the emulator can have during emulation. As an example, you can use the QEMU emulator to emulate an Android device. If you emulate an Android x86_64 device on an x86_64-based developer machine, the emulation will be much more efficient than if you emulate an ARM-based Android device on an x86_64-based developer machine.
With server virtualization, you are usually not really interested in the possibility of emulation. You are interested instead in encapsulating your application's server components. For instance, if a server application component starts to run amok and consume unreasonable amounts of CPU time or other resources, you don't want the entire physical machine to stop working altogether.

This can be achieved by creating a virtual machine with, perhaps, two cores on a machine with 64 cores. Only two cores would be affected by the runaway application. The same goes for memory allocation.

Container-based techniques provide similar degrees of encapsulation and control over resource allocation as virtualization techniques do. Containers do not normally provide the emulation features of virtualization, though. This is not an issue since we rarely need emulation for server applications.

The component that abstracts the underlying hardware and arbitrates hardware resources between different competing virtual machines is called a **hypervisor**. The hypervisor can run directly on the hardware, in which case it is called a bare metal hypervisor. Otherwise, it runs inside an operating system with the help of the operating system kernel.

VMware is a proprietary virtualization solution, and exists in desktop and server hypervisor variants. It is well supported and used in many organizations. The server variant changes names sometimes; currently, it's called VMware ESX, which is a bare metal hypervisor.

KVM is a virtualization solution for Linux. It runs inside a Linux host operating system. Since it is an open source solution, it is usually much cheaper than proprietary solutions since there are no licensing costs per instance and is therefore popular with organizations that have massive amounts of virtualization.

Xen is another type of virtualization which, amongst other features, has **paravirtualization**. Paravirtualization is built upon the idea that that if the guest operating system can be made to use a modified kernel, it can execute with greater efficiency. In this way, it sits somewhere between full CPU emulation, where a fully independent kernel version is used, and container-based virtualization, where the host kernel is used.

VirtualBox is an open source virtualization solution from Oracle. It is pretty popular with developers and sometimes used with server installations as well but rarely on a larger scale. Developers who use Microsoft Windows on their developer machines but want to emulate Linux server environments locally often find VirtualBox handy. Likewise, developers who use Linux on their workstations find it useful to emulate Windows machines.
What the different types of virtualization technologies have in common is that they provide APIs in order to allow the automation of virtual machine management. The libvirt API is one such API

that can be used with several different underlying hypervisors, such as KVM, QEMU, Xen, and LXC

Executing code on the client

Several of the configuration management systems described here allow you to reuse the node descriptors to execute code on matching nodes. This is sometimes convenient. For example, maybe you want to run a directory listing command on all HTTP servers facing the public Internet, perhaps for debugging purposes.

In the Puppet ecosystem, this command execution system is called Marionette Collective, or MCollective for short.

The Puppet master and Puppet agents

Puppet is a deployment solution that is very popular in larger organizations and is one of the first systems of its kind.

Puppet consists of a client/server solution, where the client nodes check in regularly with the Puppet server to see if anything needs to be updated in the local configuration.

The Puppet server is called a **Puppet master**, and there is a lot of similar wordplay in the names chosen for the various Puppet components.

Puppet provides a lot of flexibility in handling the complexity of a server farm, and as such, the tool itself is pretty complex.

This is an example scenario of a dialogue between a Puppet client and a Puppet master:

1. The Puppet client decides that it's time to check in with the Puppet master to discover any new configuration changes. This can be due to a timer or manual intervention by an operator at the client. The dialogue between the Puppet client and master is normally encrypted using SSL.
2. The Puppet client presents its credentials so that the Puppet master can know exactly which client is calling. Managing the client credentials is a separate issue.
3. The Puppet master figures out which configuration the client should have by compiling the Puppet catalogue and sending it to the client. This involves a number of mechanisms, and a particular setup doesn't need to utilize all possibilities.

    It is pretty common to have both a role-based and concrete configuration for a Puppet client. Role-based configurations can be inherited.

1. The Puppet master runs the necessary code on the client side such that the configuration matches the one decided on by the Puppet master.

In this sense, a Puppet configuration is declarative. You declare what configuration a machine should have, and Puppet figures out how to get from the current to the desired client state.

There are both pros and cons of the Puppet ecosystem:

Puppet has a large community, and there are a lot of resources on the Internet for Puppet. There are a lot of different modules, and if you don't have a really strange component to deploy, there already is, with all likelihood, an existing module written for your component that you can modify according to your needs.

Puppet requires a number of dependencies on the Puppet client machines. Sometimes, this gives rise to problems. The Puppet agent will require a Ruby runtime that sometimes needs to be ahead of the Ruby version available in your distribution's repositories. Enterprise distributions often lag behind in versions.

Puppet configurations can be complex to write and test.

Ansible

Ansible is a deployment solution that favors simplicity.

The Ansible architecture is agentless; it doesn't need a running daemon on the client side like Puppet does. Instead, the Ansible server logs in to the Ansible node and issues commands over SSH in order to install the required configuration.

While Ansible's agentless architecture does make things simpler, you need a Python interpreter installed on the Ansible nodes. Ansible is somewhat more lenient about the Python version required for its code to run than Puppet is for its Ruby code to run, so this dependence on Python being available is not a great hassle in practice.

Like Puppet and others, Ansible focuses on configuration descriptors that are idempotent. This basically means that the descriptors are declarative and the Ansible system figures out how to bring the server to the desired state. You can rerun the configuration run, and it will be safe, which is not necessarily the case for an imperative system.

Let's try out Ansible with the Docker method we discussed earlier.

We will use the williamyeh/ansible image, which has been developed for the purpose, but it should be possible to use any Ansible Docker image or different ones altogether, to which we just add Ansible later.

1. Create a Dockerfile with this statement:

```
FROM williamyeh/ansible:centos7
```

1. Build the Docker container with the following command:

```
docker build .
```

can use.

Normally, you would, of course, have a more complex Dockerfile that can add the things we need, but in this case, we are going to use the image interactively, so we will instead mount the directory with Ansible files from the host so that we can change them on the host and rerun them easily.

1. Run the container.

The following command can be used to run the container. You will need the hash from the previous build command:

docker run -v `pwd`/ansible:/ansible -it <hash> bash

Now we have a prompt, and we have Ansible available. The -v trick is to make parts of the host filesystem visible to the Docker guest container. The files will be visible in the /ansible directory in the container.

The playbook.yml file is as follows:

```
---
- hosts: localhost
vars:
http_port: 80
max_clients: 200
remote_user: root
tasks:
- name: ensure apache is at the latest version
yum: name=httpd state=latest
```

This playbook doesn't do very much, but it demonstrates some concepts of Ansible playbooks.

Now, we can try to run our Ansible playbook:

```
cd /ansible
ansible-playbook -i inventory playbook.yml --connection=local --sudo
```

The output will look like this:

```
PLAY [localhost] ***************************************************************
GATHERING FACTS ***************************************************************
ok: [localhost]

TASK: [ensure apache is at the latest version] ********************************
ok: [localhost]
PLAY RECAP ********************************************************************
localhost : ok=2 changed=0 unreachable=0 failed=0
```

Tasks are run to ensure the state we want. In this case, we want to install Apache's `httpd` using `yum`, and we want `httpd` to be the latest version.

To proceed further with our exploration, we might like to do more things, such as starting services automatically. However, here we run into a limitation with the approach of using Docker to emulate physical or virtual hosts. Docker is a container technology, after all, and not a full-blown virtualization system. In Docker's normal use case scenarios, this doesn't matter, but in our case, we need make some workarounds in order to proceed. The main problem is that the `systemd` init system requires special care to run in a container. Developers at Red Hat have worked out methods of doing this. The following is a slightly modified version of a Docker image by Vaclav Pavlin, who works with Red Hat:

```
FROM fedora
RUN yum -y update; yum clean all
RUN yum install ansible sudo
RUN systemctl mask systemd-remount-fs.service dev-hugepages.mount sys-fs-fuse-connections.mount
systemd-logind.service getty.target console-getty.service
RUN cp /usr/lib/systemd/system/dbus.service /etc/systemd/system/; sed -i 's/OOMScoreAdjust=-
900//' /etc/systemd/system/dbus.service
VOLUME ["/sys/fs/cgroup", "/run", "/tmp"]
ENV container=docker
CMD ["/usr/sbin/init"]
```

The environment variable `container` is used to tell the `systemd` init system that it runs inside a container and to behave accordingly.

We need some more arguments for `docker run` in order to enable `systemd` to work in the container:

```
docker run -it --rm -v /sys/fs/cgroup:/sys/fs/cgroup:ro -v `pwd`/ansible:/ansible <hash>
```

The container boots with systemd, and now we need to connect to the running container from a different shell:

docker exec -it <hash> bash

Phew! That was quite a lot of work just to get the container more lifelike! On the other hand, working with virtual machines, such as VirtualBox, is even more cumbersome in my opinion. The reader might, of course, decide differently.

Now, we can run a slightly more advanced Ansible playbook inside the container, as follows:

```
---
- hosts: localhost
vars:
http_port: 80
max_clients: 200
remote_user: root
tasks:
- name: ensure apache is at the latest version
yum: name=httpd state=latest
- name: write the apache config file
template: src=/srv/httpd.j2 dest=/etc/httpd.conf
notify:
- restart apache
- name: ensure apache is running (and enable it at boot)
service: name=httpd state=started enabled=yes
handlers:
- name: restart apache
service: name=httpd state=restarted
```

This example builds on the previous one, and shows you how to:

Install a package
Write a template file
Handle the running state of a service

The format is in a pretty simple YML syntax.

Deploying with SaltStack

SaltStack is a Python-based deployment solution.

There is a convenient dockerized test environment for Salt, by Jackson Cage. You can start it with the following:

**docker run -i -t --name=saltdocker_master_1 -h master -p 4505 -p 4506 **

**-p 8080 -p 8081 -e SALT_NAME=master -e SALT_USE=master **

-v `pwd`/srv/salt:/srv/salt:rw jacksoncage/salt

This will create a single container with both a Salt master and a Salt minion.

We can create a shell inside the container for our further explorations:

```
docker exec -i -t saltdocker_master_1 bash
```

We need a configuration to apply to our server. Salt calls configurations "states", or Salt states.

In our case, we want to install an Apache server with this simple Salt state:

top.sls:

base:

'*':

- webserver

webserver.sls:

apache2: # ID declaration

pkg: # state declaration

- installed # function declaration

Salt uses .yml files for its configuration files, similar to what Ansible does.

The file top.sls declares that all matching nodes should be of the type webserver. The webserver state declares that an apache2 package should be installed, and that's basically it. Please note that this will be distribution dependent. The Salt Docker test image we are using is based on Ubuntu, where the Apache web server package is called apache2. On Fedora for instance, the Apache web server package is instead simply called httpd.

Run the command once to see Salt in action, by making Salt read the Salt state and apply it locally:

salt-call --local state.highstate -l debug

The first run will be very verbose, especially since we enabled the debug flag!

Now, let's run the command again:

salt-call --local state.highstate -l debug

This will also be pretty verbose, and the output will end with this:

local:

ID: apache2

Function: pkg.installed

Result: True

Comment: Package apache2 is already installed.

Started: 22:55:36.937634

Duration: 2267.167 ms

Summary

Succeeded: 1

Failed: 0

Total states run: 1

Now, you can quit the container and restart it. This will clean the container from the Apache instance installed during the previous run.

This time, we will apply the same state but use the message queue method rather than applying the state locally:

salt-call state.highstate

This is the same command as used previously, except we omitted the –local flag. You could also try running the command again and verify that the state remains the same.

Deploying with Chef

Chef is a Ruby-based deployment system from Opscode.

It is pretty easy to try out Chef; for fun, we can do it in a Docker container so we don't pollute our host environment with our experiments:

```
docker run -it ubuntu
```

We need the curl command to proceed with downloading the chef installer:

```
apt-get -y install curl
curl -L https://www.opscode.com/chef/install.sh | bash
```

The Chef installer is built with a tool from the Chef team called **omnibus**. Our aim here is to try out a Chef tool called chef-solo. Verify that the tool is installed:

```
chef-solo -v
```

This will give output as:

```
Chef: 12.5.1
```

The point of chef-solo is to be able to run configuration scripts without the full infrastructure of the configuration system, such as the client/server setup. This type of testing environment is often useful when working with configuration systems, since it can be hard to get all the bits and pieces in working order while developing the configuration that you are going to deploy.

Chef prefers a file structure for its files, and a pre-rolled structure can be retrieved from GitHub. You can download and extract it with the following commands:

```
curl -L http://github.com/opscode/chef-repo/tarball/master -o master.tgz
tar -zxf master.tgz
mv chef-chef-repo* chef-repo


rm master.tgz
```

You will now have a suitable file structure prepared for Chef cookbooks, which looks like the following:

```
./cookbooks
./cookbooks/README.md
./data_bags
./data_bags/README.md
./environments
./environments/README.md
./README.md
./LICENSE
./roles
./roles/README.md
./chefignore
```

You will need to perform a further step to make everything work properly, telling chef where to find its cookbooks as:

```
mkdir .chef
echo "cookbook_path [ '/root/chef-repo/cookbooks' ]" > .chef/knife.rb
```

Now we can use the knife tool to create a template for a configuration, as follows:

```
knife cookbook create phpapp
```

Deploying with Docker

A recent alternative for deployment is Docker, which has several very interesting traits. We have already used Docker several times in this book.

You can make use of Docker's features for test automation purposes even if you use, for instance, Puppet or Ansible to deploy your products.

Docker's model of creating reusable containers that can be used on development machines, testing environments, and production environments is very appealing.

At the time of writing, Docker is beginning to have an impact on larger enterprises, but solutions such as Puppet are dominant.

While it is well known how to build large Puppet or Ansible server farms, it's not yet equally well known how to build large Docker-based server clusters.

There are several emerging solutions, such as these:

Docker Swarm: Docker Swarm is compatible with Docker Compose, which is appealing. Docker Swarm is maintained by the Docker community.

Kubernetes: Kubernetes is modeled after Google's Borg cluster software, which is appealing since it's a well-tested model used in-house in Google's vast data centers. Kubernetes is not the same as Borg though, which must be kept in mind. It's not clear whether Kubernetes offers scaling the same way Borg does.