

6

Testing the Code

If we are going to release our code early and often, we ought to be confident of its quality. Therefore, we need automated regression testing.

In this chapter, some frameworks for software testing are explored, such as **JUnit** for unit testing and **Selenium** for web frontend testing. We will also find out how these tests are run in our Continuous Integration server, Jenkins, thus forming the first part of our Continuous Delivery pipeline.

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.

JUnit in general and JUnit in particular

You need something that runs your tests. JUnit is a framework that lets you define unit tests in your Java code and run them.

JUnit belongs to a family of testing frameworks collectively called **xUnit**. The **SUnit** is the grandfather of this family and was designed by Kent Beck in 1998 for the Smalltalk language.

While JUnit is specific to Java, the ideas are sufficiently generic for ports to have been made in, for instance, C#. The corresponding test framework for C# is called, somewhat unimaginatively, **NUnit**. The N is derived from .NET, the name of the Microsoft software platform.

We need some of the following nomenclature before carrying on. The nomenclature is not specific to JUnit, but we will use JUnit as an example to make it easier to relate to the definitions.

- **Test runner:** A test runner runs tests that are defined by an xUnit framework. JUnit has a way to run unit tests from the command line, and Maven employs a test runner called Surefire. A test runner also collects and reports test results. In the case of Surefire, the reports are in XML format, and these reports can be further processed by other tools, particularly for visualization.
- **Test case:** A test case is the most fundamental type of test definition. How you create test cases differs a little bit among JUnit versions. In earlier versions, you inherited from a JUnit base class; in recent versions, you just need to annotate the test methods. This is better since Java doesn't support multiple inheritance and you might want to use your own inheritance hierarchies rather than the JUnit ones. By convention, Surefire also locates test classes that have the `Test` suffix in the class name.
- **Test fixtures:** A test fixture is a known state that the test cases can rely on so that the tests can have well-defined behavior. It is the responsibility of the developer to create these. A test fixture is also sometimes known as a test context.

With JUnit, you usually use the `@Before` and `@After` annotations to define test fixtures. `@Before` is, unsurprisingly, run before a test case and is used to bring up the environment. `@After` likewise restores the state if there is a need to.

Sometimes, `@Before` and `@After` are more descriptively named **Setup** and **Teardown**. Since annotations are used, the method can have the names that are the most intuitive in that context.

- **Test suites:** You can group test cases together in test suites. A test suite is usually a set of test cases that share the same test fixture.
- **Test execution:** A test execution runs the tests suites and test cases.

Here, all the previous aspects are combined. The test suites and test cases are located, the appropriate test fixtures are created, and the test cases run. Lastly, the test results are collected and collated.

- **Test result formatter:** A test result formatter formats test result output for human consumption. The format employed by JUnit is versatile enough to be used by other testing frameworks and formatters not directly associated with JUnit. So, if you have some tests that don't really use any of the xUnit frameworks, you can still benefit by presenting the test results in Jenkins by providing a test result XML file. Since the file format is XML, you can produce it from your own tool, if need be.
- **Assertions:** An assertion is a construct in the xUnit framework that makes sure that a condition is met. If it is not met, it is considered an error, and a test error is reported. The test case is also usually terminated when the assertion fails.

JUnit has a number of assertion methods available. Here is a sample of the available assertion methods:

- To check whether two objects are equal:
`assertEquals(str1, str2);`
- To check whether a condition is true:
`assertTrue (val1 < val2);`
- To check whether a condition is false:
`assertFalse(val1 > val2);`

A JUnit example

JUnit is well supported by Java build tools. It will serve well as an example of JUnit testing frameworks in general.

If we use Maven, by convention, it will expect to find test cases in the following directory:

```
/src/test/java
```

Mocking

Mocking refers to the practice of writing simulated resources to enable unit testing. Sometimes, the words "fake" or "stub" are used. For example, a middleware system that responds with JSON structures from a database would "mock" the database backend for its unit tests. Otherwise, the unit tests would require the database backend to be online, probably also requiring exclusive access. This wouldn't be convenient.

Mockito is a mocking framework for Java that has also been ported to Python.

Test Coverage

When you hear people talk about unit testing, they often talk about test coverage. Test coverage is the percentage of the application code base that is actually executed by the test cases.

In order to measure unit test code coverage, you need to execute the tests and keep track of the code that has or hasn't been executed.

Cobertura is a test coverage measurement utility for Java that does this. Other such utilities include jcoverage and Clover.

Cobertura works by instrumenting the Java bytecode, inserting code fragments of its own into already compiled code. These code fragments are executed while measuring code coverage during execution of test cases

It's usually assumed that a hundred percent test coverage is the ideal. This might not always be the case, and one should be aware of the cost/benefit trade-offs.

A simple counterexample is a simple getter method in Java:

```
private int positiveValue;
void setPositiveValue(int x){
    this.positiveValue=x;
}

int getPositiveValue(){
    return positiveValue;
}
```

If we write a test case for this method, we will achieve a higher test coverage. On the other hand, we haven't achieved much of anything, in practice. The only thing we are really testing is that our Java implementation doesn't have bugs.

If, on the other hand, the setter is changed to include validation to check that the value is not a negative number, the situation changes. As soon as a method includes logic of some kind, unit testing is useful.

Automated integration testing

Automated integration testing is similar in many ways to unit testing with respect to the basic techniques that are used. You can use the same test runners and build system support. The primary difference with unit testing is that less mocking is involved.

Where a unit test would simply mock the data returned from a backend database, an integration test would use a real database for its tests. A database is a decent example of the kind of testing resources you need and what types of problems they could present.

Automated integration testing can be quite tricky, and you need to be careful with your choices.

If you are testing, say, a read-only middleware adapter, such as a SOAP adapter for a database, it might be possible to use a production database copy for your testing. You need the database contents to be predictable and repeatable; otherwise, it will be hard to write and run your tests.

The added value here is that we are using a production data copy. It might contain data that is hard to predict if you were to create test data from scratch. The requirements are the same as for manual testing. With automated integration testing, you need, well, more automation than with manual testing. For databases, this doesn't have to be very complicated. Automated database backup and restore are well-known operations.

Docker in automated testing

Docker can be quite convenient when building automated test rigs. It adds some of the features of unit testing but at a functional level. If your application consists of several server components in a cluster, you can simulate the entire cluster with a set of containers. Docker provides a virtual network for the cluster that makes clear how the containers interact at the network level.

Docker also makes it easy to restore a container to a known state. If you have your test database in a Docker container, you can easily restore the database to the same state as before the tests taking place. This is similar to restoring the environment in the `After` method in unit tests.

The Jenkins Continuous Integration server has support for the starting and stopping of containers, which can be useful when working with Docker test automation.

Using Docker Compose to run the containers you need is also a useful option.

Docker is still young, and some aspects of using Docker for test automation can require glue code that is less than elegant.

A simple example could be firing up a database container and an application server container that communicate with each other. The basic process of starting the containers is simple and can be done with a shell script or Docker Compose. But, since we want to run tests on the application server container that has been started, how do we know whether it was properly started? In the case of the WildFly container, there is no obvious way to determine the running state apart from watching the log output for occurrences of strings or maybe polling a web socket. In any case, these types of hacks are not very elegant and are time consuming to write. The end result can be well worth the effort, though.

Arquillian

Arquillian is an example of a test tool that allows a level of testing closer to integration testing than unit testing together with mocking allows. Arquillian is specific to Java application servers such as WildFly. Arquillian is interesting because it illustrates the struggle of reaching a closer approximation of production systems during testing. You can reach such approximations in any number of ways, and the road to there is filled with trade-offs.

There is a "hello world" style demonstration of Arquillian in the book's source code archive.

Performance testing

Performance testing is an essential part of the development of, for instance, large public web sites.

Performance testing presents similar challenges as integration testing. We need a testing system that is similar to a production system in order for the performance test data to be useful to make a forecast about real production system performance.

The most commonly used performance test is load testing. With load testing, we measure, among other things, the response time of a server while the performance testing software generates synthetic requests for the server.

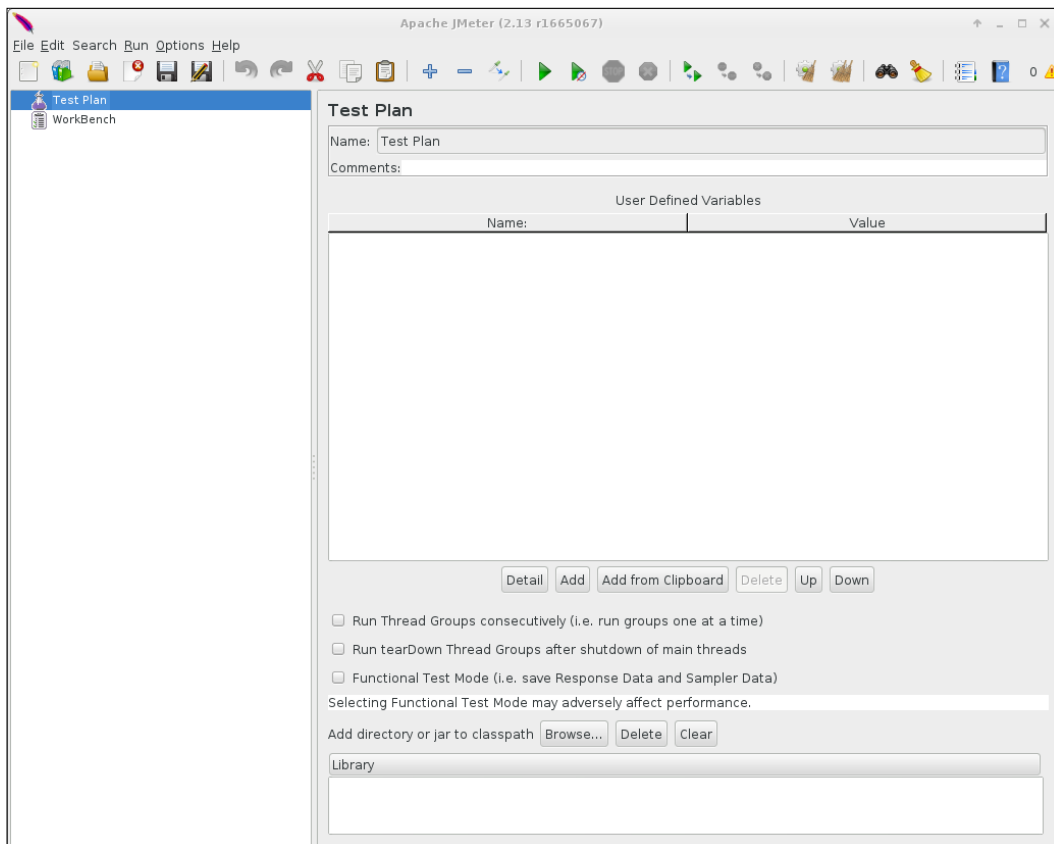
Apache JMeter is an example of an open source application for measuring performance. While it's simpler than its proprietary counterparts, such as LoadRunner, JMeter is quite useful, and simplicity is not really a bad thing.

JMeter can generate simulated load and measure response times for a number of protocols, such as HT, LDAP, SOAP, and JDBC.

There is a JMeter Maven plugin, so you can run JMeter as part of your build.

JMeter can also be used in a Continuous Integration server. There is a plugin for Jenkins, called the performance plugin, that can execute JMeter test scenarios.

Ideally, the Continuous Integration server will deploy code that has been built to a test environment that is production-like. After deployment, the performance tests will be executed and test data collected, as shown in this screenshot:



Automated acceptance testing

Automated acceptance testing is a method of ensuring that your testing is valid from the end user's point of view.

Cucumber is a framework where test cases are written in plaintext and associated with test code. This is called **behavior-driven development**. The original implementation of Cucumber was written in Ruby, but ports now exist for many different languages.

The appeal of Cucumber from a DevOps point of view is that it is intended to bring different roles together. Cucumber feature definitions are written in a conversational style that can be achieved without programming skills. The hard data required for test runs is then extracted from the descriptions and used for the tests.

While the intentions are good, there are difficulties in implementing Cucumber that might not immediately be apparent. While the language of the behavior specifications is basically free text, they still need to be somewhat spartan and formalized; otherwise, it becomes difficult to write matching code that extracts the test data from the descriptions. This makes writing the specifications less attractive to the roles that were supposed to write them in the first place. What then happens is that programmers write the specifications, and they often dislike the verbosity and resort to writing ordinary unit tests.

As with many things, cooperation is of the essence here. Cucumber can work great when developers and product owners work together on writing the specifications in a way that works for everyone concerned.

Now, let's look at a small "hello world" style example for Cucumber.

Cucumber works with plaintext files called feature files, which look like this:

```
Feature: Addition
  I would like to add numbers with my pocket calculator

  Scenario: Integer numbers
    * I have entered 4 into the calculator
    * I press add
    * I have entered 2 into the calculator
    * I press equal
    * The result should be 6 on the screen
```

The feature description is implementation-language neutral. Describing the Cucumber test code is done in a vocabulary called **Gherkin**.

If you use the Java 8 lambda version of Cucumber, a test step could look somewhat like this:

```
Calculator calc;
public MyStepdefs() {
    Given("I have entered (\\d+) into the calculator", (Integer i) -> {
        System.out.format("Number entered: %n\\n", i);
        calc.push(i);
    });
}
```

```
When("I press (\\w+)", (String op) -> {  
    System.out.format("operator entered: %n\\n", op);  
    calc.op(op);  
});  
Then("The result should be (\\d+)", (Integer i) -> {  
    System.out.format("result : %n\\n", i);  
    assertThat(calc.result(), i);  
});
```

The complete code can, as usual, be found in the book's source archive.

This is a simple example, but it should immediately be apparent both what the strengths and weaknesses with Cucumber are. The feature descriptions have a nice human-readable flair. However, you have to match the strings with regular expressions in your test code. If your feature description changes even slightly in wording, you will have to adjust the test code.

Automated GUI testing

Automating GUI testing has many desirable properties, but it is also difficult. One reason is that user interfaces tend to change a lot during the development phase, and buttons and controls move around in the GUI.

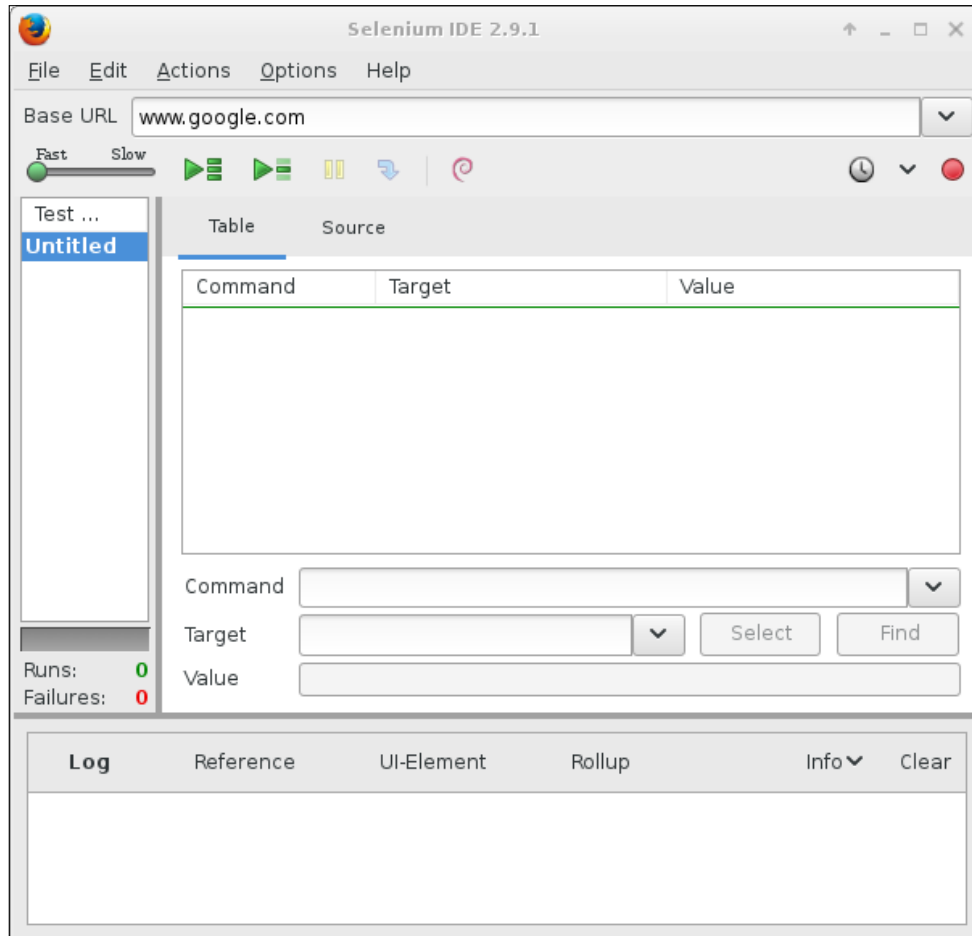
Older generations of GUI testing tools often worked by synthesizing mouse events and sending them to the GUI. When a button moved, the simulated mouse click went nowhere, and the test failed. It then became expensive to keep the tests updated with changes in the GUI.

Selenium is a web UI testing toolkit that uses a different, more effective, approach. The controllers are instrumented with identifiers so that Selenium can find the controllers by examining the document object model (DOM) rather than blindly generating mouse clicks.

Selenium works pretty well in practice and has evolved over the years.

Another method is employed by the Sikuli test framework. It uses a computer vision framework, OpenCV, to help identify controllers even if they move or change appearances. This is useful for testing native applications, such as games.

The screenshot included below is from the Selenium IDE.



Integrating Selenium tests in Jenkins

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.

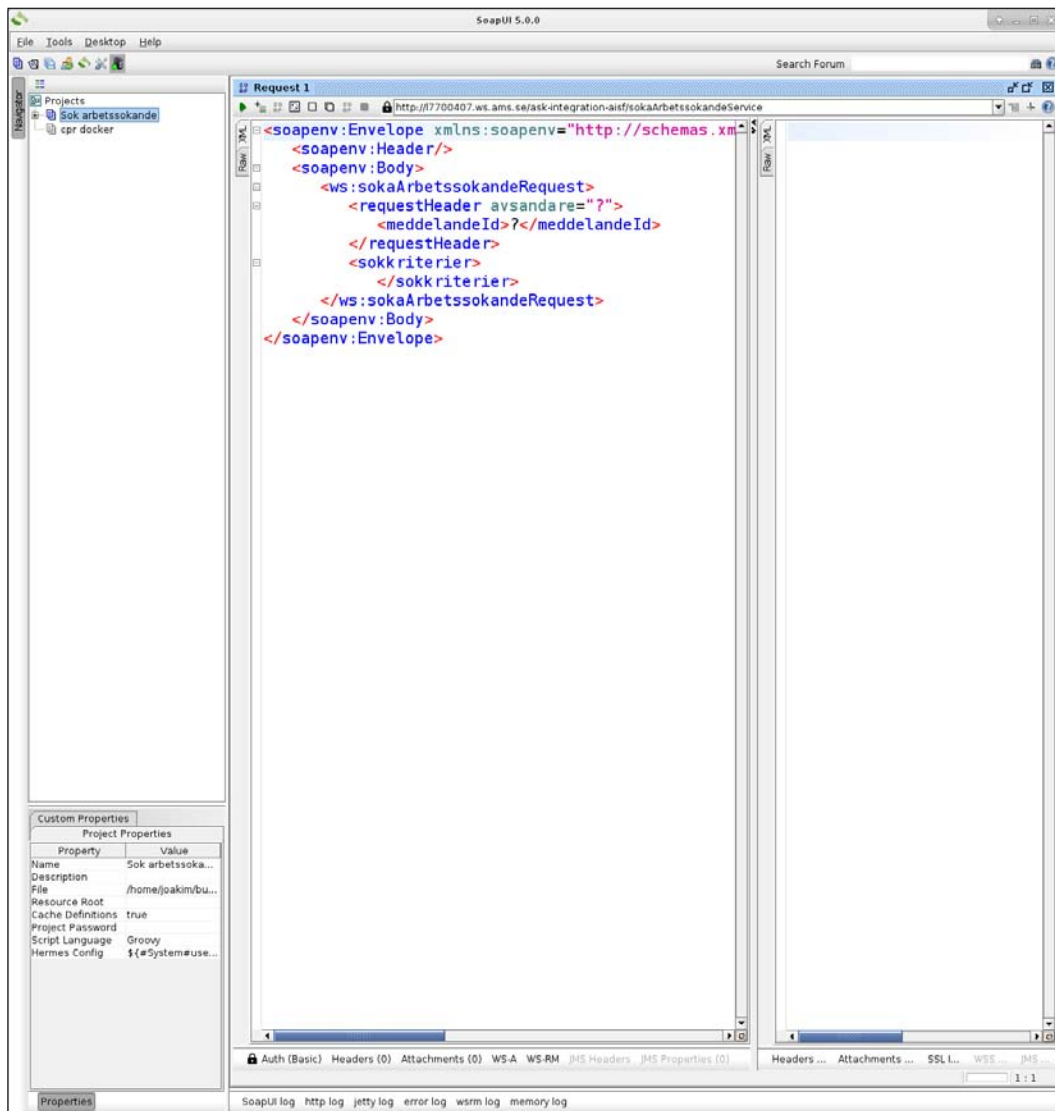
The soapUI is a good example of a tool that appeals to several different roles. Testers who build test cases get a fairly well-structured environment for writing tests and running them interactively. Tests can be built incrementally.

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.

A complete test automation scenario

We have looked at a number of different ways of working with test automation. Assembling the pieces into a cohesive whole can be daunting.

In this section, we will have a look at a complete test automation example, continuing from the user database web application for our organization, Matangle.

You can find the source code in the accompanying source code bundle for the book.

The application consists of the following layers:

- A web frontend
- A JSON/REST service interface
- An application backend layer
- A database layer

The test code will work through the following phases during execution:

- Unit testing of the backend code
- Functional testing of the web frontend, performed with the Selenium web testing framework
- Functional testing of the JSON/REST interface, executed with soapUI

All the tests are run in sequence, and when all of them succeed, the result can be used as the basis for a decision to see whether the application stack is deemed healthy enough to deploy to a test environment, where manual testing can commence.

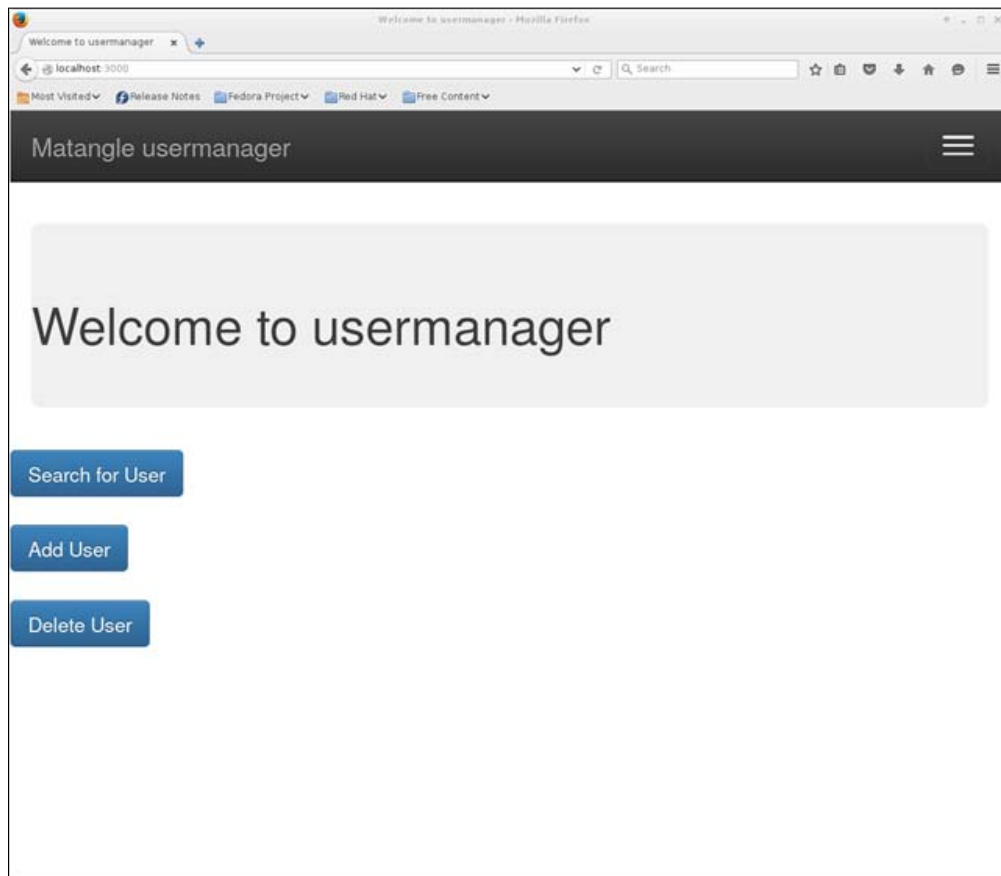
Manually testing our web application

Before we can automate something usefully, we need to understand the details of what we will be automating. We need some form of a test plan.

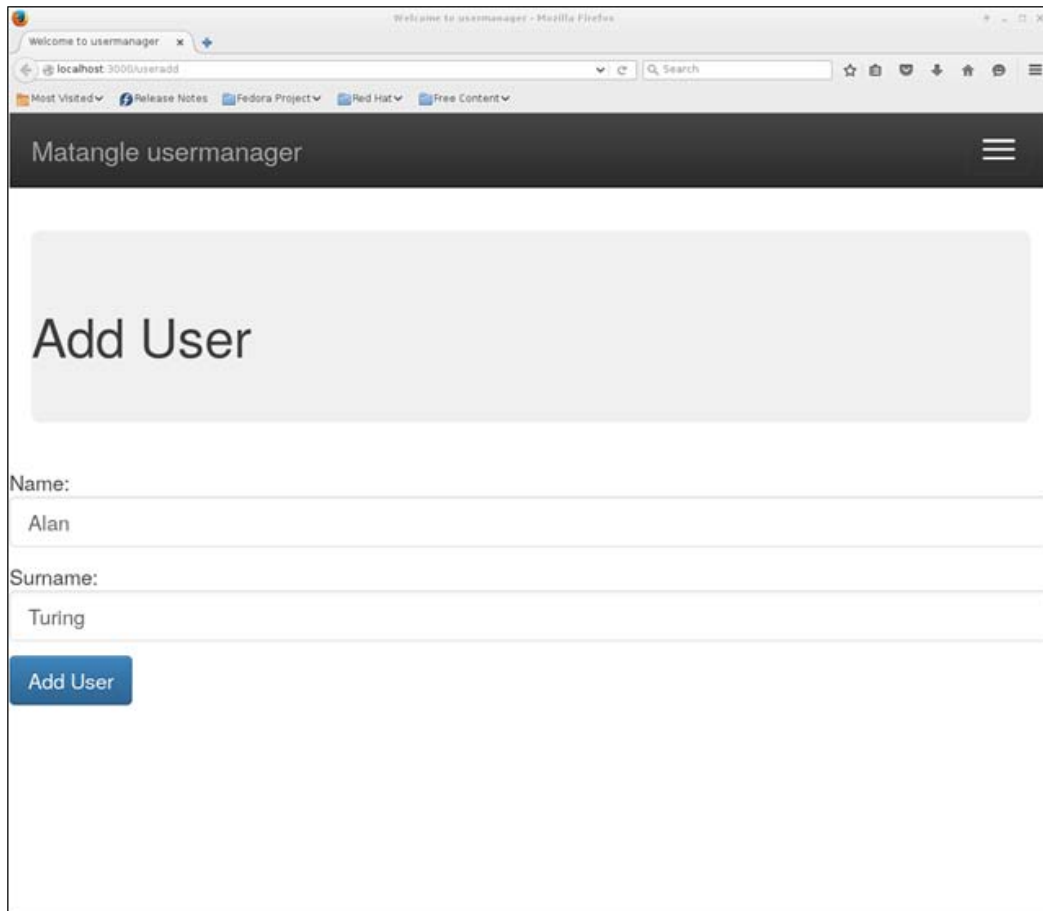
Below, we have a test plan for our web application. It details the steps that a human tester needs to perform by hand if no test automation is available. It is similar to what a real test plan would look like, except a real plan would normally have many more formalities surrounding it. In our case, we will go directly to the details of the test in question:

1. Start a fresh test. This resets the database backend to a known state and sets up the testing scenario so that manual testing can proceed from a known state.

The tester points a browser to the application's starting URL:



2. Click on the **Add User** link.
3. Add a user:

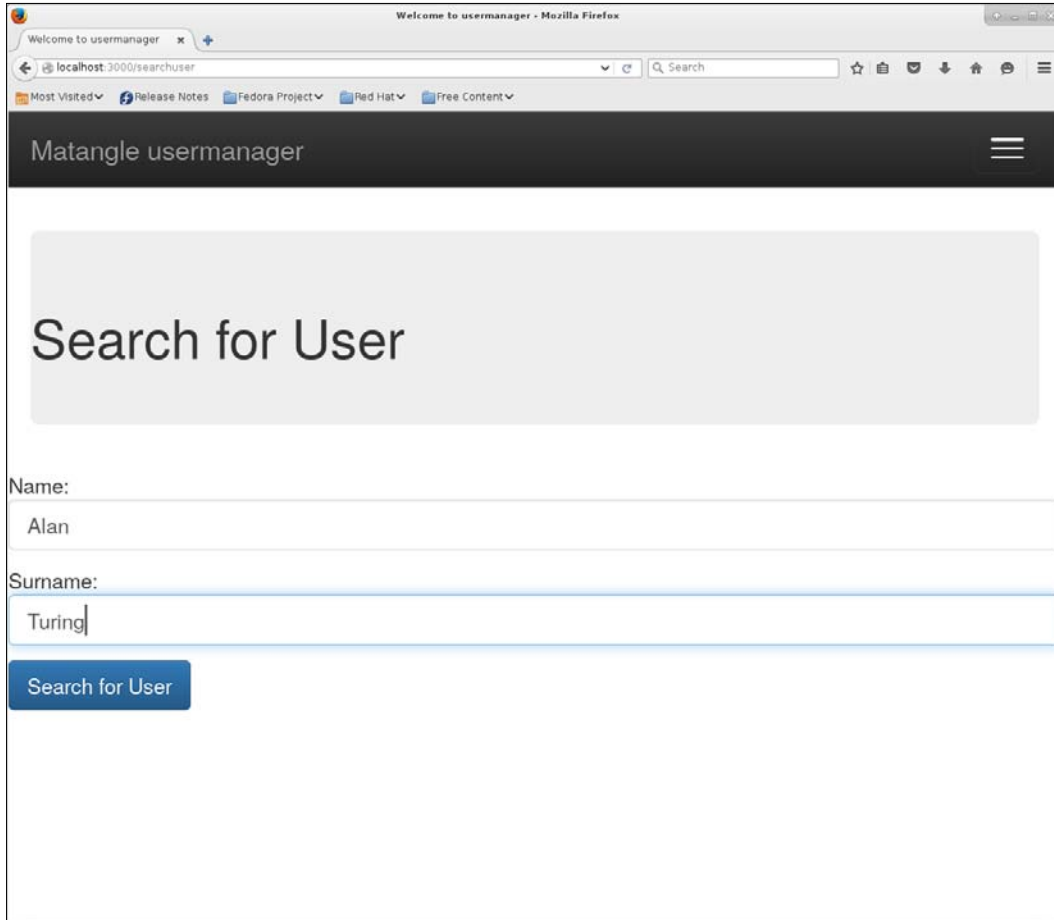


The screenshot shows a web browser window with the address bar at `localhost:3000/useradd`. The page title is "Welcome to usermanager - Mozilla Firefox". The browser's Most Visited list includes "Release Notes", "Fedora Project", "Red Hat", and "Free Content". The application header is "Matangle usermanager" with a hamburger menu icon. The main content area has a large grey box with the text "Add User". Below this, there are two text input fields: "Name:" with the value "Alan" and "Surname:" with the value "Turing". At the bottom left of the form is a blue button labeled "Add User".

Enter a username—Alan Turing, in our test case.

4. Save the new user. A success page will be shown.

5. Verify that the user was added properly by performing a search:



The screenshot shows a web browser window with the title "Welcome to usermanager - Mozilla Firefox". The address bar shows "localhost:3000/searchuser". The browser's bookmark bar includes "Most Visited", "Release Notes", "Fedora Project", "Red Hat", and "Free Content". The application header is "Matangle usermanager" with a hamburger menu icon on the right. The main content area has a large grey box with the text "Search for User". Below this, there are two input fields: "Name:" with the value "Alan" and "Surname:" with the value "Turing". A blue button labeled "Search for User" is positioned below the input fields.

Click on the **Search User** link. Search for **Alan Turing**. Verify that **Alan** is present in the result list.

While the reader is probably less than impressed with the application's complexity at this point, this is the level of detail we need to work with if we are going to be able to automate the scenario, and it is this complexity that we are studying here.

Running the automated test

The test is available in a number of flavors in the source bundle.

To run the first one, you need a Firefox installation.

Choose the one called `autotest_v1`, and run it from the command line:

```
autotest_v1/bin/autotest.sh
```

If all goes well, you will see a Firefox window open, and the test you previously performed by hand will be done automatically. The values you filled in and the links you clicked on by hand will all be automated.

This isn't foolproof yet, because maybe the Firefox version you installed isn't compatible, or something else is amiss with the dependencies. The natural reaction to problems like these is dependency management, and we will look at a variant of dependency management using Docker shortly.

Finding a bug

Now we will introduce a bug and let the test automation system find it.

As an exercise, find the string "Turing" in the test sources. Change one of the occurrences to "Tring" or some other typographical error. Just change one; otherwise, the verification code will believe there is no error and that everything is alright!

Run the tests again, and notice that the error is found by the automatic test system.

Test walkthrough

Now we have run the tests and verified that they work. We have also verified that they are able to discover the bug we created.

What does the implementation look like? There is a lot of code, and it would not be useful to reprint it in the book. It is useful, though, to give an overview of the code and have a look at some snippets of the code.

Open the `autotest_v1/test/pom.xml` file. It's a Maven project object model file, and it's here that all the plugins used by the tests are set up. Maven POM files are declarative XML files and the test steps are step-by-step imperative instructions, so in the latter case, Java is used.

There's a property block at the top, where dependency versions are kept. There is no real need to break out the versions; it has been used in this case to make the rest of the POM file less version-dependent:

```
<properties>
  <junit.version>XXX</junit.version>
  <selenium.version>XXX</selenium.version>
  <cucumber.version>XXX</cucumber.version>
  ...
</properties>
```

Here are the dependencies for JUnit, Selenium and Cucumber:

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>${junit.version}</version>
</dependency>

<dependency>
  <groupId>org.seleniumhq.selenium</groupId>
  <artifactId>selenium-java</artifactId>
  <version>${selenium.version}</version>
</dependency>

<dependency>
  <groupId>info.cukes</groupId>
  <artifactId>cucumber-core</artifactId>
  <version>${cucumber.version}</version>
</dependency>

<dependency>
  <groupId>info.cukes</groupId>
  <artifactId>cucumber-java</artifactId>
  <version>${cucumber.version}</version>
</dependency>

<dependency>
  <groupId>info.cukes</groupId>
  <artifactId>cucumber-junit</artifactId>
  <version>${cucumber.version}</version>
</dependency>
```


To define the tests according to the Cucumber method, we need a feature file that describes the test steps in a human-readable language. This feature file corresponds to our previous test plan for manual tests:

```
Feature: Manage users
  As an Administrator
  I want to be able to
  - Create a user
  - Search for the user
  - Delete the user

Scenario: Create a named user
  Given a user with the name 'Alan'
  And the surname 'Turing'
  When the administrator clicks 'Add User'
  Then the user should be added

Scenario: Search for a named user
  Given a user with the name 'Alan'
  And the surname 'Turing'
  When the administrator clicks 'Search for User'
  Then the user 'Alan Turing' should be shown

Scenario: Delete a named user
  Given a user with the name 'Alan'
  And the surname 'Turing'
  When the administrator clicks 'Delete User'
  Then the user should be deleted
```

The feature file is mostly plaintext with small elements of machine-readable markup. It's up to the corresponding test code to parse the plaintext of the scenarios with regexes.

It is also possible to localize the feature files to the language used within your own team. This can be useful since the feature files might be written by people who are not accustomed to English.

The feature needs actual concrete code to execute, so you need some way to bind the feature to the code.

You need a test class with some annotations to make Cucumber and JUnit work together:

```
@RunWith(Cucumber.class)
@Cucumber.Options(
    glue = "matangle.glue.manageUser",
    features = "features/manageUser.feature",
    format = {"pretty", "html:target/Cucumber"}
)
```

In this example, the names of the Cucumber test classes have, by convention, a Step suffix.

Now, you need to bind the test methods to the feature scenarios and need some way to pick out the arguments to the test methods from the feature description. With the Java Cucumber implementation, this is mostly done with Java annotations. These annotations correspond to the keywords used in the feature file:

```
@Given("."+a user with the name '(.+)')")
public void addUser(String name) {
```

In this case, the different inputs are stored in member variables until the entire user interface transaction is ready to go. The sequence of operations is determined by the order in which they appear in the feature file.

To illustrate that Cucumber can have different implementations, there is also a Clojure example in the book's source code bundle.

So far, we have seen that we need a couple of libraries for Selenium and Cucumber to run the tests and how the Cucumber feature descriptor is bound to methods in our test code classes.

The next step is to examine how Cucumber tests execute Selenium test code.

Cucumber test steps mostly call classes with Selenium implementation details in classes with a View suffix. This isn't a technical necessity, but it makes the test step classes more readable, since the particulars of the Selenium framework are kept in a separate class.

The Selenium framework takes care of the communication between the test code and the browser. View classes are an abstraction of the web page that we are automating. There are member variables in the view code that correspond to HTML controllers. You can describe the binding between test code member variables and HTML elements with annotations from the Selenium framework, as follows:

```
@FindBy(id = "name") private WebElement nameInput;
@FindBy(id = "surname") private WebElement surnameInput;
```

The member variable is then used by the test code to automate the same steps that the human tester followed using the test plan. The partitioning of classes into view and step classes also makes the similarity of the step classes to a test plan more apparent. This separation of concerns is useful when people involved with testing and quality assurance work with the code.

To send a string, you use a method to simulate a user typing on a keyboard:

```
nameInput.clear();  
nameInput.sendKeys(value);
```

There are a number of useful methods, such as `click()`, which will simulate a user clicking on the control.

Handling tricky dependencies with Docker

Because we used Maven in our test code example, it handled all code dependencies except the browser. While you could clearly deploy a browser such as Firefox to a Maven-compatible repository and handle the test dependency that way if you put your mind to it, this is normally not the way this issue is handled in the case of browsers. Browsers are finicky creatures and show wildly differing behavior in different versions. We need a mechanism to run many different browsers of many different versions.

Luckily, there is such a mechanism, called Selenium Grid. Since Selenium has a pluggable driver architecture, you can essentially layer the browser backend in a client server architecture.

To use Selenium Grid, you must first determine how you want the server part to run. While the easiest option would be to use an online provider, for didactic reasons, it is not the option we will explore here.

There is an `autotest_seleniumgrid` directory, which contains a wrapper to run the test using Docker in order to start a local Selenium Grid. You can try out the example by running the wrapper.

The latest information regarding how to run Selenium Grid is available on the project's GitHub page.

Selenium Grid has a layered architecture, and to set it up, you need three parts:

- A `RemoteWebDriver` instance in your testing code. This will be the interface to Selenium Grid.
- Selenium Hub, which can be seen as a proxy for browser instances.

- Firefox or Chrome grid nodes. These are the browser instances that will be proxied by Hub.

The code to set up the `RemoteWebDriver` could look like this:

```
DesiredCapabilities capabilities = new DesiredCapabilities();
capabilities.setPlatform(Platform.LINUX);
capabilities.setBrowserName("Firefox");
    capabilities.setVersion("35");
    driver = new RemoteWebDriver(
        new URL("http://localhost:4444"),
        capabilities);
```

The code asks for a browser instance with a particular set of capabilities. The system will do its best to oblige.

The code can only work if there is a Selenium Grid Hub running with a Firefox node attached.

Here is how you start Selenium Hub using its Docker packaging:

```
docker run -d -p 4444:4444 --name selenium-hub selenium/hub
```

And here is how you can start a Firefox node and attach it to Hub:

```
docker run -d --link selenium-hub:hub selenium/node-firefox
```

Summary

This concludes the test code walkthrough. When you read the code, you might want to use only a subset of the ideas that are illustrated. Maybe the Cucumber method isn't really a good fit for you, or you value concise and succinct code over the layered abstractions used in the example. That is a natural and sound reaction. Adopt the ideas so they work for your team. Also, have a look at the other flavors of the testing code available in the source bundle when deciding on what works for you!

Software testing is a vast subject that can fill volumes. In this chapter, we surveyed some of the many different types of testing available. We also looked at concrete ways of working with automated software testing in a Continuous Integration server. We used Jenkins and Maven as well as JUnit and JMeter. Although these tools are Java oriented, the concepts translate readily to other environments.

Now that we have built and tested our code, we will start working with deploying our code in the next chapter.

7

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.

We have a typical enterprise application, with a number of different high-level 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.

Configuring the base OS

The configuration of the base operating system must be dealt with somehow.

Often, our application stack has subtle, or not so subtle, requirements on the base operating system. Some application stacks, such as Java, Python, or Ruby, make these operating system requirements less apparent, because these technologies go to a great length to offer cross-platform functionality. At other times, the operating system requirements are apparent to a greater degree, such as when you work with low-level mixed hardware and software integrations, which is common in the telecom industry.

There are many existing solutions that deal with this fundamental issue. Some systems work with a bare metal (or bare virtual machine) approach, where they install the desired operating system from scratch and then install all the base dependencies that the organization needs for their servers. Such systems include, for example, Red Hat Satellite and Cobbler, which works in a similar way but is more lightweight.

Cobbler allows you to boot a physical or virtual machine over the network using `dhcpd`. The DHCP server can then allow you to provide a netboot-compliant image. When the netboot image is started, it contacts Cobbler to retrieve the packages that will be installed in order to create the new operating system. Which packages are installed can be decided on the server from the target machine's network MAC address for instance.

Another method that is very popular today is to provide base operating system images that can be reused between machines. Cloud systems such as AWS, Azure, or OpenStack work this way. When you ask the cloud system for a new virtual machine, it is created using an existing image as a base. Container systems such as Docker also work in a similar way, where you declare your base container image and then describe the changes you want to formulate for your own image.

Describing clusters

There must be a way to describe clusters.

If your organization only has a single machine with a single application, then you might not need to describe how a cluster deployment of your application would look like. Unfortunately (or fortunately, depending on your outlook), the reality is normally that your applications are spread out over a set of machines, virtual or physical.

All the systems we work with in this chapter support this idea in different ways. Puppet has an extensive system that allows machines to have different roles that in turn imply a set of packages and configurations. Ansible and Salt have these systems as well. The container-based Docker system has an emerging infrastructure for describing sets of containers connected together and Docker hosts that can accept and deploy such cluster descriptors.

Cloud systems such as AWS also have methods and descriptors for cluster deployments.

Cluster descriptors are normally also used to describe the application layer.

Delivering packages to a system

There must be a way to deliver packages to a system.

Much of an application can be installed as packages, which are installed unmodified on the target system by the configuration management system. Package systems such as RPM and deb have useful features, such as verifying that the files provided by a package are not tampered with on a target system, by providing checksums for all files in the package. This is useful for security reasons as well as debugging purposes. Package delivery is usually done with operating system facilities such as yum package channels on Red Hat based systems, but sometimes, the configuration management system can also deliver packages and files with its own facilities. These facilities are often used in tandem with the operating system's package channels.

There must be a way to manage configurations that is independent of installed packages.

The configuration management system should, obviously, be able to manage our applications' configurations. This is complex because configuration methods vary wildly between applications, regardless of the many efforts that have been made to unify the field.

The most common and flexible system to configure applications relies on text-based configuration files. There are several other methods, such as using an application that provides an API to handle configuration (such as a command-line interface) or sometimes handling configuration via database settings.

In my experience, configuration systems based on text files create the least amount of hassle and should be preferred for in-house code at least. There are many ways to manage text-based configurations. You can manage them in source code handling systems such as Git. There's a host of tools that can ease the debugging of broken configuration, such as `diff`. If you are in a tight spot, you can edit configurations directly on the servers using a remote text editor such as Emacs or Vi.

Handling configurations via databases is much less flexible. This is arguably an anti-pattern that usually occurs in organizations where the psychological rift between developer teams and operations teams are too wide, which is something we aim to solve with DevOps. Handling configurations in databases makes the application stack harder to get running. You need a working database to even start the application.

Managing configuration settings via imperative command-line APIs is also a dubious practice for similar reasons but can sometimes be helpful, especially if the API is used to manage an underlying text-based configuration. Many of the configuration management systems, such as Puppet, depend on being able to manage declarative configurations. If we manage the configuration state via other mechanisms, such as command-line imperative API, Puppet loses many of its benefits.

Even managing text-based configuration files can be a hassle. There are many ways for applications to invent their own configuration file formats, but there are a set of base file formats that are popular. Such file formats include XML, YML, JSON, and INI.

Usually, configuration files are not static, because if they were, you could just deploy them with your package system like any piece of binary artifact.

Normally, the application configuration files need to be based on some kind of template file that is later instantiated into a form suitable for the machine where the application is being deployed.

An example might be an application's database connector descriptor. If you are deploying your application to a test environment, you want the connector descriptor to point to a test database server. Vice versa, if you are deploying to a production server, you want your connector to point to a production database server.

As an aside, some organizations try to handle this situation by managing their DNS servers, such that an example database DNS alias `database.yourorg.com` resolves to different servers depending on the environment. The domain `yourorg.com` should be replaced with your organization's details of course, and the database server as well.

Being able to use different DNS resolvers depending on the environment is a useful strategy. It can be difficult for a developer, however, to use the equivalent mechanism on his or her own development machine. Running a private DNS server on a development machine can be difficult, and managing a local host file can also prove cumbersome. In these cases, it might be simpler to make the application have configurable settings for database hosts and other backend systems at the application level.

Many times, it is possible to ignore the details of the actual configuration file format altogether and just rely on the configuration system's template managing system. These usually work by having a special syntax for placeholders that will be replaced by the configuration management system when creating the concrete configuration file for a concrete server, where the application will be deployed. You can use the exact same basic idea for all text-based configuration files, and sometimes even for binary files, even though such hacks should be avoided if at all possible.

The XML format has tools and infrastructure that can be useful in managing configurations, and XML is indeed a popular format for configuration files. For instance, there is a special language, XSLT, to transform XML from one structural form to another. This is very helpful in some cases but used less in practice than one might expect. The simple template macro substitution approach gets you surprisingly far and has the added benefit of being applicable on nearly all text-based configuration formats. XML is also fairly verbose, which also makes it unpopular in some circles. YML can be seen as a reaction to XML's verbosity and can accomplish much of the same things as XML, with less typing.

Another useful feature of some text configuration systems that deserves mention is the idea that a base configuration file can include other configuration files. An example of this is the standard Unix `sudo` tool, which has its base configuration in the `/etc/sudoers` file, but which allows for local customization by including all the files that have been installed in the directory `/etc/sudoers.d`.

This is very convenient, because you can provide a new `sudoer` file without worrying too much about the existing configuration. This allows for a greater degree of modularization, and it is a convenient pattern when the application allows it.

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.

A note about the exercises

It is pretty easy to try out the various deployment systems using Docker to manage the base operating system, where we will do our experiments. It is a time-saving method that can be used when developing and debugging the deployment code specific to a particular deployment system. This code will then be used for deployments on physical or virtual machines.

We will first try each of the different deployment systems that are usually possible in the local deployment modes. Further down the line, we will see how we can simulate the complete deployment of a system with several containers that together form a virtual cluster.

We will try to use the official Docker images if possible, but sometimes there are none, and sometimes the official image vanishes, as happened with the official Ansible image. Such is life in the fast-moving world of DevOps, for better or for worse.

It should be noted, however, that Docker has some limitations when it comes to emulating a full operating system. Sometimes, a container must run in elevated privilege modes. We will deal with those issues when they arise.

It should also be noted that many people prefer Vagrant for these types of tests. I prefer to use Docker when possible, because it's lightweight, fast, and sufficient most of the time.



Keep in mind that actually deploying systems in production will require more attention to security and other details than we provide here.

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.

4. 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
```
2. Build the Docker container with the following command:

```
docker build .
```

This will download the image and create an empty Docker container that we 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.

3. 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 YAML syntax.

PalletOps

PalletOps is an advanced deployment system, which combines the declarative power of Lisp with a very lightweight server configuration.

PalletOps takes Ansible's agentless idea one step further. Rather than needing a Ruby or Python interpreter installed on the node that is to be configured, you only need `ssh` and a `bash` installation. These are pretty simple requirements.

PalletOps compiles its Lisp-defined DSL to Bash code that is executed on the slave node. These are such simple requirements that you can use it on very small and simple servers—even phones!

On the other hand, while there are a number of support modules for Pallet called **crates**, there are fewer of them than there are for Puppet or Ansible.

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 -zxvf 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 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 .yaml 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
```

Changes:

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:

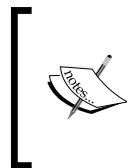
```
salt-call state.highstate
```

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.

Salt versus Ansible versus Puppet versus PuppetOps execution models

While the configuration systems we explore in this chapter share a fair number of similarities, they differ a lot in the way code is executed on the client nodes:

- With Puppet, a Puppet agent registers with the Puppet master and opens a communication channel to retrieve commands. This process is repeated periodically, normally every thirty minutes.



Thirty minutes isn't fast. You can, of course, configure a lower value for the time interval required for the next run. At any rate, Puppet essentially uses a pull model. Clients must check in to know whether changes are available.

- Ansible pushes changes over SSH when desired. This is a push model.
- Salt uses a push model, but with a different implementation. It employs a ZeroMQ messaging server that the clients connect to and listen for notifications about changes. This works a bit like Puppet, but faster.

Which method is best is an area of contention between developer communities. Proponents of the message queue architecture believe that it is faster and that speed matters. Proponents of the plain SSH method claim that it is fast enough and that simplicity matters. I lean toward the latter stance. Things tend to break, and the likelihood of breakage increases with complexity.

Vagrant

Vagrant is a configuration system for virtual machines. It is geared towards creating virtual machines for developers, but it can be used for other purposes as well.

Vagrant supports several virtualization providers, and VirtualBox is a popular provider for developers.

First, some preparation. Install `vagrant` according to the instructions for your distribution. For Fedora, the command is this:

```
yum install 'vagrant*'
```

This will install a number of packages. However, as we are installing this on Fedora, we will experience some problems. The Fedora Vagrant packages use `libvirt` as a virtual machine provider rather than VirtualBox. That is useful in many cases, but in this case, we would like to use VirtualBox as a provider, which requires some extra steps on Fedora. If you use some other distribution, the case might be different.

First, add the VirtualBox repository to your Fedora installation. Then we can install VirtualBox with the `dnf` command, as follows:

```
dnf install VirtualBox
```

VirtualBox is not quite ready to be used yet, though. It needs special kernel modules to work, since it needs to arbitrate access to low-level resources. The VirtualBox kernel driver is not distributed with the Linux kernel. Managing Linux kernel drivers outside of the Linux source tree has always been somewhat inconvenient compared to the ease of using kernel drivers that are always installed by default. The VirtualBox kernel driver can be installed as a source module that needs to be compiled. This process can be automated to a degree with the `dkms` command, which will recompile the driver as needed when there is a new kernel installed. The other method, which is easier and less error-prone, is to use a kernel module compiled for your kernel by your distribution. If your distribution provides a kernel module, it should be loaded automatically. Otherwise, you could try `modprobe vboxdrv`. For some distributions, you can compile the driver by calling an `init.d` script as follows:

```
sudo /etc/init.d/vboxdrv setup
```

Now that the Vagrant dependencies are installed, we can bring up a Vagrant virtual machine.

The following command will create a Vagrant configuration file from a template. We will be able to change this file later. The base image will be `hashicorp/precise32`, which in turn is based on Ubuntu.

```
vagrant init hashicorp/precise32
```

Now, we can start the machine:

```
vagrant up
```

If all went well, we should have a `vagrant` virtual machine instance running now, but since it is headless, we won't see anything.

Vagrant shares some similarities with Docker. Docker uses base images that can be extended. Vagrant also allows this. In the Vagrant vocabulary, a base image is called a **box**.

To connect to the headless `vagrant` instance we started previously, we can use this command:

```
vagrant ssh
```

Now we have an `ssh` session, where we can work with the virtual machine. For this to work, Vagrant has taken care of a couple of tasks, such as setting up keys for the SSH communication channel for us.

Vagrant also provides a configuration system so that Vagrant machine descriptors can be used to recreate a virtual machine that is completely configured from source code.

Here is the Vagrantfile we got from the earlier stage. Comments are removed for brevity.

```
Vagrant.configure(2) do |config|
  config.vm.box = "hashicorp/precise32"
end
```

Add a line to the Vagrantfile that will call the bash script that we will provide:

```
Vagrant.configure("2") do |config|
  config.vm.box = "hashicorp/precise32"
  config.vm.provision :shell, path: "bootstrap.sh"
end
```

The `bootstrap.sh` script will look like this:

```
#!/usr/bin/env bash
apt-get update
apt-get install -y apache2
```

This will install an Apache web server in the Vagrant-managed virtual machine.

Now we know enough about Vagrant to be able to reason about it from a DevOps perspective:

- Vagrant is a convenient way of managing configurations primarily for virtual machines based on VirtualBox. It's great for testing.
- The configuration method doesn't really scale up to clusters, and it's not the intended use case, either.
- On the other hand, several configuration systems such as Ansible support Vagrant, so Vagrant can be very useful while testing our configuration code.

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.

Comparison tables

Everyone likes coming up with new words for old concepts. While the different concepts in various products don't always match, it's tempting to make a dictionary that maps the configuration systems' different terminology with each other.

Here is such a terminology comparison chart:

System	Puppet	Ansible	Pallet	Salt
Client	Agent	Node	Node	Minion
Server	Master	Server	Server	Master
Configuration	Catalog	Playbook	Crate	Salt State

Also, here is a technology comparison chart:

System	Puppet	Ansible	Pallet	Chef	Salt
Agentless	No	Yes	Yes	Yes	Both
Client dependencies	Ruby	Python, sshd, bash	sshd, bash	Ruby, sshd, bash	Python
Language	Ruby	Python	Clojure	Ruby	Python

Cloud solutions

First, we must take a step back and have a look at the landscape. We can either use a cloud provider, such as AWS or Azure, or we can use our own internal cloud solution, such as VMware or OpenStack. There are valid arguments for both external and internal cloud providers or even both, depending on your organization.

Some types of organizations, such as government agencies, must store all data regarding citizens within their own walls. Such organizations can't use external cloud providers and services and must instead build their own internal cloud equivalents.

Smaller private organizations might benefit from using an external cloud provider but can't perhaps afford having all their resources with such a provider. They might opt to have in-house servers for normal loads and scale out to an external cloud provider during peak loads.

Many of the configuration systems we have described here support the management of cloud nodes as well as local nodes. PalletOps supports AWS and Puppet supports Azure, for instance. Ansible supports a host of different cloud services.

AWS

Amazon Web Services allows us to deploy virtual machine images on Amazon's clusters. You can also deploy Docker images. Follow these steps to set up AWS:

1. Sign up for an account with AWS. Registration is free of charge, but a credit card number is required even for the free of charge tier
2. Some identity verification will need to happen, which can be done via an automated challenge-response phone call.
3. When the user verification process is complete, you will be able to log in to AWS and use the web console.



In my opinion, the AWS web console does not represent the epitome of web interface usability, but it gets the job done. There are a host of options, and in our case, we are interested in the virtual machine and Docker container options.

4. Go to **EC2 network and security**. Here you can create management keys that will be required later.

As a first example, let's create the default container example provided by AWS, console-sample-app-static. To log in to the generated server, you need first to create an SSH key pair and upload your public key to AWS. Click through all the steps and you will get a small sample cluster. The final resource creation step can be slow, so it's the perfect opportunity to grab a cup of coffee!

5. Now, we can view the details of the cluster and choose the web server container. You can see the IP address. Try opening it in a web browser.

Now that we have a working account on AWS, we can manage it with the configuration management system of our choosing.

Azure

Azure is a cloud platform from Microsoft. It can host both Linux and Microsoft virtual machines. While AWS is the service people often default to, at least in the Linux space, it never hurts to explore the options. Azure is one such option that is gaining market share at the moment.

Creating a virtual machine on Azure for evaluation purposes is comparable to creating a virtual machine on AWS. The process is fairly smooth.

Summary

In this chapter, we explored some of the many options available to us when deploying the code we built. There are a lot many options, and that is for a reason. Deployment is a difficult subject, and you will likely spend a lot of time figuring out which option suits you best.

In the next chapter, we will explore the topic of monitoring our running code.