Agile DevOps: Test-driven infrastructure

Continually testing scripted environments

Paul Duvall

November 06, 2012

Few in the software industry question that writing automated tests for application code is a good practice. Teams are now applying similar automated testing practices to infrastructure and environments. In this Agile DevOps installment, DevOps expert Paul Duvall covers writing automated tests for your infrastructure using tools such as Cucumber with Gherkin. These tests can be run in conjunction with every scripted change to the infrastructure to ensure quick feedback when a change introduces an error into an environment.

When provisioning environments, most organizations apply antipatterns that reduce the reliability and repeatability of creating those environments:

- Some teams manually install and configure resources required to create an environment. The problem is that it usually takes days, weeks, or months to get an environment into the desired state. Over this period of time, team members might come and go or forget the precise steps, making it impossible for any one person to re-create the environments as intended.
- Other teams try to document each and every step for provisioning environments. Although this is a noble effort and much better than not documenting the process, the documentation is quickly outdated, incorrect, or not run as specified (that is, subject to human error). For these and other reasons, I do not recommend documentation as the source of record for creating environments (although, as you'll see in the next section, it can be a first step in a systematic approach to provisioning and testing environments).
- Some teams have been “burned” by previous teams that manually installed or documented everything. Determined to learn from their mistakes, they automate the infrastructure provisioning using a collection of scripts. Months later, while mired in myriad custom shell and other scripts, they decide that the environments proved to be too difficult to automate successfully.

About this series

Developers can learn a lot from operations, and operations can learn a lot from developers. This series of articles is dedicated to exploring the practical uses of applying an operations
In contrast, some teams — after reading about companies embracing DevOps — have learned from the mistakes of manual provisioning, document-driven provisioning, and the collection of disjointed scripts. They escape these antipatterns by using a test-driven approach coupled with the use of mature infrastructure automation tools, as part of a single path to production. As I mentioned in the "Infrastructure automation" article in this series, these teams treat the infrastructure as code. In this article, you’ll learn how to take a test-driven approach to defining infrastructure.

Principles and practices

On numerous occasions, I’ve written about a five-step heuristic (see Related topics) — five key steps to making a process continuous. They are: document, test, script, version, and (make the process) continuous. I describe each of these steps in greater detail here in the context of an infrastructure, where all the steps work in tandem:

- **Document**: Document the steps for performing a process. For an infrastructure, you’ll document the type and version of components such as operating system, application and web servers, and databases. You’ll also describe in detail how to install, configure, and run these infrastructure components. When using this approach, the key is to document to automate, because you will eventually dispose of the documentation.

- **Test**: Write an automated test that describes the intended outcome. For an infrastructure, you can define features or expected outcomes such as a server running in a certain location, or the presence of a certain file or directory.

- **Script**: Script all of the actions for the process. For an infrastructure, you’ll use tools such as Chef or Puppet to define these environments according to the tests specified in the preceding step.

- **Version**: Version source files. For an infrastructure, these source files are defined in infrastructure automation scripts such as those created with Puppet or Chef.

- **Continuous**: The process must be scripted to the point at which it can run in a headless manner — humans aren’t required to run the commands. For an infrastructure, teams configure a Continuous Integration (CI) server to run infrastructure automation scripts and scripted-environment tests as part of a CI job — meaning that the infrastructure is rebuilt with every source file change.

Get involved

developerWorks Agile transformation provides news, discussions, and training to help you and your organization build a foundation on agile development principles.

The distinctive characteristic of infrastructure tests is the order of execution. Although you can still use a test-first approach, the infrastructure scripts apply the entire infrastructure; then, the tests are run to verify the efficacy of the infrastructure automation. This sequence is in contrast to application-code unit testing, in which a test is run, followed by the execution of a specific method or set of methods. Essentially, no atomicity is involved in a test-driven infrastructure —
the entire environment is created, and then the suite of tests is run against the newly provisioned environment.

The most common approach to applying test-driven infrastructure is to start by documenting the steps for provisioning an infrastructure. Based on this documentation, an engineer writes some automated tests that describe the expected outcomes of the infrastructure. These tests might include what the team expects to be installed when an environment is fully provisioned. Then, an engineer writes scripts and commits them to a version-control system. Finally, the scripts and automated tests are run as part of a CI system. Some combination of these steps — even if it's often not the "textbook" sequential process I've laid out — ensures a robust test-driven infrastructure that provides quick feedback to all team members.

**How it works**

One of the many ways to implement a test-driven infrastructure is to use a behavior-driven development (BDD) approach. When you use BDD, you define requirements and tests in the same file, called a **feature file**. Next I'll show you some examples of feature files written in the Cucumber and Gherkin domain-specific languages (DSLs).

Listing 1 shows a Cucumber feature file — called production.feature — that describes a feature, its background, and its scenarios:

### Listing 1. Defining executable specification in Cucumber and Gherkin

```
Feature: Scripted Provisioning of Target Environment
As a Developer
I would like my target environment provisioned correctly
so I can deploy applications to it

Background:
Given I am sshed into the environment

Scenario: Is the proper version of Postgresql installed?
When I run "/usr/bin/postgres --version"
Then I should see "8.4"

Scenario: Is the proper version of Apache installed?
When I run "/usr/sbin/httpd -v"
Then I should see "2.2"

Scenario: Is the proper version of Java installed?
When I run "java -version"
Then I should see "1.6"

Scenario: Is the proper version of perl installed?
When I run "perl -version"
Then I should see "perl, v5.10.1"

Scenario: Is the proper version of make installed?
When I run "make -version"
Then I should see "GNU Make 3.81"

Scenario: Is the proper version of Ruby installed?
When I run "ruby -v"
Then I should see "ruby 1.9.3"

Scenario: Is the proper version of GCC installed?
```
When I run "gcc -v"
Then I should see "4.4.6"

Feature, Background, Given, Scenario, When, and Then are all Cucumber-specific words that are part of its DSL for defining feature files. The Scripted Provisioning of Target Environment feature in Listing 1 describes the expected outcome in a ubiquitous language that all team members — business analysts, developers, testers, operations and so forth — use in an organization. Specific actions such as /usr/bin/postgres --version utilize Gherkin — a dependent tool that is installed when you install Cucumber. This BDD approach provides a way to define an executable specification that can be run through automated tools.

Listing 2 shows a Cucumber feature file that describes a way to specify that a particular major version of a tool is installed. Under this approach, the infrastructure script won't fail if it encounters any major version of this particular tool.

### Listing 2. Cucumber feature checking for major tool version

Feature: Scripted Provisioning of Target Environment
As a Developer
I would like my target environment provisioned correctly so I can deploy applications to it

Background:
Given I am sshed into the environment

Scenario: Is Tomcat installed?
When I check the version of Tomcat installed
Then the major version should be 6

In Listing 3, the lines following the Example keyword indicate that the behavior described in the scenario has multiple successful outcomes.

### Listing 3. Describing examples in Cucumber and Gherkin

Feature: Scripted Provisioning of Target Environment
As a Developer
I would like my target environment provisioned correctly so I can deploy applications to it

Background:
Given I am sshed into the environment

Scenario Outline: Is the proper version of libxml2-devel installed?
When I run "sudo yum info libxml2-devel"
Then I should see "<output>"

Examples: output I should see
| output |
| Installed Packages |
| 2.7.6 |

**Looser coupling**

The coupling between test and infrastructure is not as tight as it is between unit tests (such as those written with JUnit) and application code. For example (and as a reminder of how you might
script an environment), Listing 4 is an example of a script (or *manifest*) that uses the Puppet infrastructure automation tool:

**Listing 4. Puppet manifest that describes an httpd server called **httpd**

```puppet
class httpd {
    package { 'httpd-devel':
        ensure => installed,
    }
    service { 'httpd':
        ensure => running,
        enable => true,
        subscribe => Package['httpd-devel'],
    }
}
```

Notice that unlike an application-code unit test, the test in Listing 1 doesn't execute this Puppet manifest. Instead, the configuration defined in the manifest (along with many others) is first applied to an infrastructure to create an environment. Then, a suite of tests like those defined in the Listings 1, 2, and 3 is run to verify that the environment is in the expected state, as defined by those tests.

**Static analysis**

At the time of this writing, the ecosystem for static analysis tools in infrastructure automation is rather sparse. A few of the same tools that you use for static analysis of application code are suitable, but others aren't ready for the peculiarities of infrastructure scripts — mainly because infrastructure scripts aren't procedural or object-oriented in nature. Most are declarative, meaning that developers determine how the environment is defined in the script; the script is responsible for how it actually applies the configuration using the infrastructure.

Tools such as Simian are useful for identifying sections of similar code. And foodcritic is a lint-like tool for the open source infrastructure automation tool, Chef. For coverage — because tools like Chef and Puppet don't use a procedural or object-oriented language — an effective approach is to define "requirements coverage" using BDD tools such as Cucumber rather than the traditional code-coverage tools you might apply to application code.

**Test-driven everything**

In this article, you learned that infrastructure can be tested much like application code — or any other part of the software system. You saw examples of implementing test-driven infrastructure using the Cucumber and supporting Gherkin BDD languages. Finally, you learned that — although static analysis for infrastructure differs from static analysis for application code — you can use some static analysis tools to help you improve infrastructure automation code.

In the next article, you'll learn about versioning all components of a software system: infrastructure, application code, configuration, and data.
Related topics

- "Test Driven Infrastructure with Vagrant, Puppet and Guard": In this SysAdvent blog entry, Patrick Debois writes about test-driven infrastructures using Vagrant, Puppet, and Guard.
- **Cucumber**: Learn more about BDD with Cucumber.
- **IBM Tivoli® Provisioning Manager**: Tivoli Provisioning Manager enables a dynamic infrastructure by automating the management of physical servers, virtual servers, software, storage, and networks.
- **cucumber-nagios**: cucumber-nagios lets you use a natural language to describe how a system should work.
- **Foodcritic**: A lint-like tool for your Opscode Chef cookbooks.
- **IBM Tivoli System Automation for Multiplatforms**: Tivoli System Automation for Multiplatforms provides high availability and automation for enterprise-wide applications and IT services.
- **Evaluate IBM products** in the way that suits you best: Download a product trial, try a product online, or use a product in a cloud environment.

© Copyright IBM Corporation 2012

Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)