SAMPLE CHAPTER

Re-Engineering LEGACY SOFTWARE

Chris Birchall

MANNING

Re-Engineering Legacy Software
by Chris Birchall

Chapter 7

brief contents

7 Automating the development environment

This chapter covers

- The value of a good README file
- Automating the development environment using Vagrant and Ansible
- Removing dependencies on external resources to make developers more autonomous

Nearly all software has some dependencies on its surrounding environment. The software (or rather the humans who developed it) expects certain other software to be running and some configuration to have been performed on the machine where it will run. For example, a lot of applications use a database to store their data, so they depend on a database server running somewhere and being accessible. As an example of a configuration dependency, the software might expect a certain directory to exist so it can save its log files inside it.

Discovering all of these dependencies, and provisioning the environment in order to satisfy them, can be quite tedious. The dependencies are often poorly documented, simply because there's no good place or standard format in which to document them.

In this chapter we'll look at how to make it as easy as possible for you and others to set up a development environment and get started on maintaining a piece of legacy software. We'll write scripts that not only automate the provisioning process, but also act as documentation, making it easy for later maintainers to understand the software's dependencies.

7.1 First day on the job

Congratulations, and welcome to your new job at Fzzle, Inc. Once the HR guy has given you the tour of the office, he takes you to meet your new boss. Anna is the tech lead of the User Services team, where you'll be working as a full-stack developer. She shows you your desk and fills you in on the details of the job.

First, a little background. Fzzle is a survivor of the dot-com bubble, a veteran of the social web sector. At its heart, it's a social network, but over the years it has accumulated a large number of auxiliary services and microsites. With a business model based on a combination of freemium membership and targeted ads, the company enjoys steady growth and boasts a few million users.

Fzzle.com's architecture is service-oriented, with a few dozen services of varying size, age, and implementation technology making up the site. As the vagueness of the User Services team's name suggests, the team is in charge of developing and maintaining a variety of these services. Unfortunately, there aren't enough developers in the team to properly maintain all the services they own, so Anna is really happy to have another pair of hands to share the workload. Currently she and the other developers are busy writing a brand new service, so you'll be tasked with maintaining some of the legacy services that they haven't had time to look after recently.

Your first job will be to add a new feature to the User Activity Dashboard (UAD). This is an in-house web application designed to be used by Fzzle's marketing department and advertising partners. It displays detailed information about how many users have been active on the site, what they've been viewing, which user segments are growing, and so on, so that advertisers can plan targeted ad campaigns. Unfortunately the UAD is quite old and hasn't received much development love recently, despite being critical to the business.

Anna tells you to start by cloning the Git repository and getting the UAD running on your local machine. She tells you to ask for help if you get stuck, but she looks really busy, so you decide to try your best to get things set up on your own, without bothering her. After all, how hard can it be?

7.1.1 Setting up the UAD development environment

As instructed, you clone the repository and take a look around. First stop, the README file. Hmm, no README? That's odd. I guess you'll have to do some detective work to find out how to build and run this thing.

Next you find an Ant build file, some Java source files, and a web.xml file. Now you're getting somewhere. It looks like it's a standard Java web app, so it should run

on any web app container. You have experience with Apache Tomcat, so you grab a copy from the website and install it. You also head over to the Oracle website and download the latest Java package.

Once you've got Java and Tomcat installed, you open up the Ant build file in your editor and work out how to compile the application and package it into a WAR file, suitable for deployment on Tomcat. You find a target called `package` that seems to do what you want, so you go to the Ant website, download and install Ant, and then run `ant package`.

```
$ ant package
Buildfile: /Users/chris/code/uad/build.xml

clean:

compile:
    [mkdir] Created dir: /Users/chris/code/uad/dest
    [javac] Compiling 157 source files to /Users/chris/code/uad/dest

package:
      [war] Building war: /Users/chris/code/uad/uad.war

BUILD SUCCESSFUL
Total time: 15 seconds
```

Hey presto, it worked! You copy the resulting WAR file to your Tomcat webapps directory, start Tomcat, and point your browser to http://localhost:8080/uad/.

Unfortunately, you're met with a completely blank page. Checking the Tomcat logs, you find the following error message:

```
Cannot start the User Activity Dashboard. Make sure $RESIN_3_HOME is set.
```

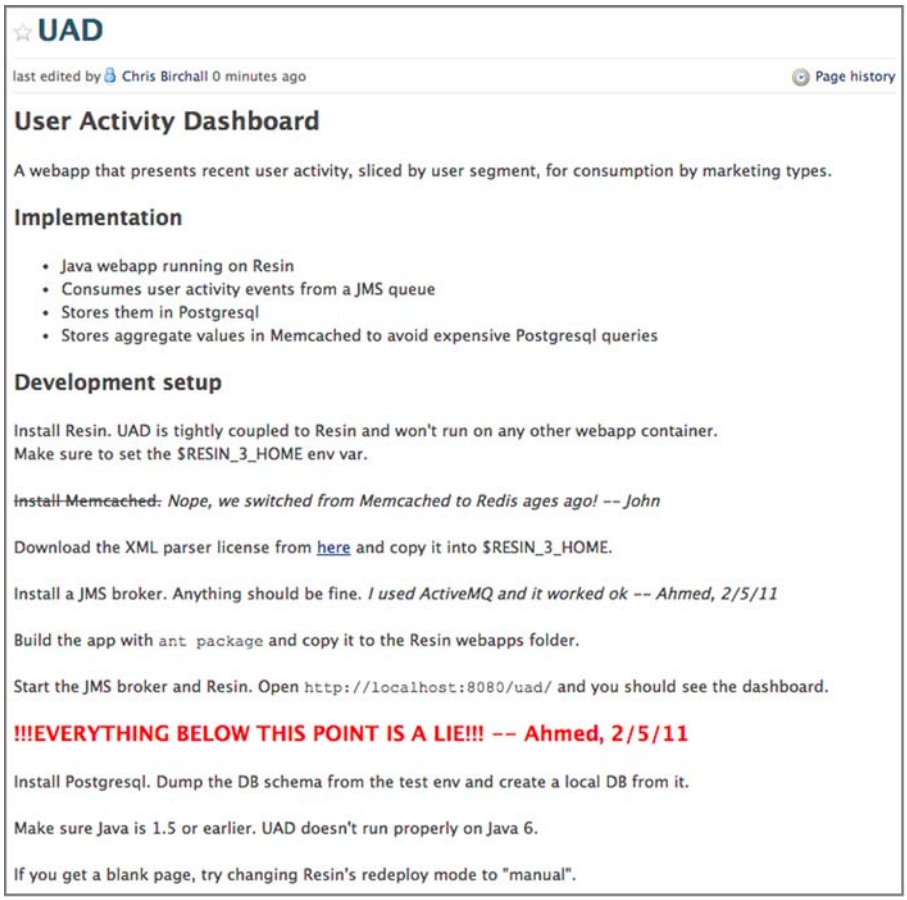
Resin? You've heard of that, but never used it before. It's a Java web application container, just like Tomcat. Based on the error message, it looks like you need to install version 3. You head over to the Resin website, find a download page for Resin 3 (which turns out to be very old and deprecated, but luckily still available), download and install it. You do your best to configure it based on information you find in a blog post.

After copying the WAR file to the Resin webapps folder and starting Resin, you're immediately greeted with another error message:

```
Failed to connect to DB (jdbc:postgresql://testdb/uad)
Check DB username and password.
```

While you're ruminating on this error message, Anna comes over and says, "Sorry, I forgot to tell you, the instructions to set up UAD are on the developer wiki. Take a look at http://devwiki/ and you should be able to find it."

The wiki page, shown in figure 7.1, clears up a few mysteries. But it also appears to be unreliable and poorly maintained, so anything written on that page should probably be taken with a grain of salt.



UAD

last edited by Chris Birchall 0 minutes ago — Page history

User Activity Dashboard

A webapp that presents recent user activity, sliced by user segment, for consumption by marketing types.

Implementation

- Java webapp running on Resin
- Consumes user activity events from a JMS queue
- Stores them in Postgresql
- Stores aggregate values in Memcached to avoid expensive Postgresql queries

Development setup

Install Resin. UAD is tightly coupled to Resin and won't run on any other webapp container.
Make sure to set the $RESIN_3_HOME env var.

~~Install Memcached.~~ *Nope, we switched from Memcached to Redis ages ago! -- John*

Download the XML parser license from here and copy it into $RESIN_3_HOME.

Install a JMS broker. Anything should be fine. *I used ActiveMQ and it worked ok -- Ahmed, 2/5/11*

Build the app with `ant package` and copy it to the Resin webapps folder.

Start the JMS broker and Resin. Open `http://localhost:8080/uad/` and you should see the dashboard.

!!!EVERYTHING BELOW THIS POINT IS A LIE!!! -- Ahmed, 2/5/11

Install Postgresql. Dump the DB schema from the test env and create a local DB from it.

Make sure Java is 1.5 or earlier. UAD doesn't run properly on Java 6.

If you get a blank page, try changing Resin's redeploy mode to "manual".

Figure 7.1 The developer wiki page for the User Activity Dashboard

Based on the wiki page, it looks like the application has a number of external dependencies, some of which you have already discovered.

- It needs a Java web app container, specifically Resin 3.
- It needs a Java Messaging Service (JMS) message broker to provide a message queue. It looks like Apache ActiveMQ is a good bet.
- It needs a PostgreSQL database in which to store its raw data.
- It needs a Redis instance in which to store aggregate data.
- Finally, it needs the license file for a proprietary XML parser to be installed.

You make a note of this and go back to trying to solve your DB connection issue. Looking at the error message in the logs again, it looks like the application is trying to connect to a PostgreSQL DB in the test environment. But where is it getting that JDBC URL from? There must be a configuration file somewhere. Looking in the Git repository again, you find a file called config.properties:

```
# Developer config for User Activity Dashboard.
# These values will be overridden with environment vars in production.

db.url=jdbc:postgresql://testdb/uad
# If you don't have a DB user, ask the ops team to create one
db.username=Put your DB username here
db.password=Put your DB password here

redis.host=localhost
redis.port=6379

jms.host=localhost
jms.port=61616
jms.queue=uad_messages
```

Following the instructions in the configuration file comment, you fire off an email to the ops team asking for a DB account for the test environment. By this time it's already past 3 p.m., so you're unlikely to get a response today.

In the meantime, you can get started on installing and configuring Redis and ActiveMQ, and working out where to put that XML parser license key ...

7.1.2 What went wrong?

That story got pretty long, but I wanted you to feel the frustration of the detective work that's often needed when starting work on a legacy codebase for the first time.

There were a few distinct, but interrelated, reasons why the experience of setting up the UAD project was so painful. Let's look at them in turn.

Poor documentation

The first problem with the documentation for the UAD project was that it wasn't discoverable. There's no point in writing documentation if people can't find it.

In general, the closer the documentation is to the source code, the easier it is for a developer to find. I recommend storing the documentation inside the same repository as the source code, preferably in the root directory where it's easy to spot. If you prefer to keep your documentation elsewhere, such as on a wiki, then at least add a text file to the repository with a link to the docs.

The second problem is that the wiki page appeared to have no proper structure. This meant that not only was it difficult to read, it was also difficult to update. There was no policy for how to update it, so over time it gradually become a mishmash of hints, corrections, and out-of-date or dubious information.

The key to making documentation easy to write, easy to read, and easy to update is pretty simple: give it some structure, and keep it short. We'll take a look at how to do this in the next section.

Lack of automation

As you probably noticed, there were a lot of tedious manual steps involved in getting the UAD working on a development machine. A lot of these steps were quite similar, something like this:

1. Download something.
2. Install it (unzip it and copy it somewhere).
3. Configure it (update some values in a text file).

This sort of procedure is just crying out to be automated. The benefits of automation are manifold:

- *Less developer time wasted*—Remember John and Ahmed who left notes on the wiki page? They, along with who knows how many others, had to go through exactly the same manual steps as you did.
- *Less verbose documentation*—This means it's more likely to be read and more likely to be kept up to date.
- *Better parity between developers' machines*—All developers will be running exactly the same versions of the same software.

RELIANCE ON EXTERNAL RESOURCES

You had to ask the ops team to create a PostgreSQL user for you. This sort of thing (a step that involves waiting for a human to respond) can really slow down the onboarding process, and it can be very frustrating to wait for somebody to reply to your email before you can finish setting up your development environment.

Ideally a developer should be able to install and run all necessary dependencies on their local machine. That way they have complete control over those dependencies. If they need to create a database user, for example, they can go ahead and do it. (Or, even better, have a script do it for them!) Doing everything you can to maintain developers' autonomy and remove blocking steps in workflows can really boost productivity.

Having developers run everything locally also reduces the scope for people to tread on each other's toes. You'll no longer hear developers saying things like, "Anybody mind if I change the time on this server? I need to test something" or "Oops, I just wiped the DB in the test environment. Sorry everyone!"

In the rest of the chapter we'll look at how to improve the onboarding process for UAD, first by improving the documentation and then by adding some automation.

7.2 The value of a good README

In my experience, a README file in the root folder of the source code's repository is the most effective form of documentation. It's highly discoverable, it's likely to remain up to date because it's close to the source code, and if written well it will make a new developer's onboarding process fast and painless.

Another bonus is that, because it's in the same repository as the source code, the README becomes a target for code review. Whenever you make a change to the software that alters the development environment's setup procedure, the reviewer can check that the README has been updated accordingly.

README FILE FORMAT The README should be a human-readable plain text file so that developers can open it in their editor of choice. There are a number of

popular formats for structured text, but my preference is for Markdown. I find it easy to read and write, especially when including code samples, and sites such as GitHub can render it beautifully.

A README file should be structured as follows.

Listing 7.1 An example README file in Markdown format

```
# My example software

Brief explanation of what the software is.

## Dependencies

* Java 7 or newer
* Memcached
...

The following environment variables must be set when running:

* `JAVA_HOME`
...

## How to configure

Edit `conf/dev-config.properties` if you want to change the Memcached port, etc.

## How to run locally

1. Make sure Memcached is running
2. Run `mvn jetty:run`
3. Point your browser at `http://localhost:8080/foo`

## How to run tests

1. Make sure Memcached is running
2. Run `mvn test`

## How to build

Run `mvn package` to create a WAR file in the `target` folder.

## How to release/deploy

Run `./release.sh` and follow the prompts.
```

This example gives a new developer exactly the information they need to get started, and nothing more. The key to keeping the README file useful is to make it concise.

You may want to write more documentation about your software, such as to explain the architecture or how to troubleshoot when things go wrong in production, but this should not go into the README. You could write additional documentation on a wiki (making sure to link to it from the README) or perhaps add a docs folder to the repository and write it in separate Markdown files there.

Of course, if a lot of manual setup is needed to get the software running locally, it will be difficult to keep the README short. In the next section we'll look at how to automate the environment setup, which will solve this problem.

7.3 Automating the development environment with Vagrant and Ansible

There are a variety of tools available to help you automate the setup of a development machine. In the remainder of this chapter we'll use Vagrant and Ansible to perform this automation for the UAD project.

We'll look at both tools in detail soon, but here's a quick preview of what they do and why we want to use them.

- *Vagrant*—Vagrant automates the process of managing virtual machines (VMs), either on your local development machine or in the cloud. The point of this is that you can have one VM for each piece of software that you develop. The software's dependencies (Ruby runtimes, databases, web servers, and so on) are all kept inside the VM, so they're nicely isolated from everything else you have installed on your machine.
- *Ansible*—Ansible automates the provisioning of your application— the installation and configuration of all of its dependencies. You write down the steps needed in a set of YAML files, and Ansible takes care of performing those steps. This automation makes provisioning easy and repeatable and reduces the chance of incorrect provisioning due to human error.

7.3.1 Introducing Vagrant

Vagrant is a tool that allows you to programmatically build an isolated environment for your application and all of its dependencies.

The Vagrant environment is a VM, so it enjoys complete isolation from both the host machine and any other Vagrant machines you may be running. For the underlying VM technology, Vagrant supports VirtualBox, VMware, and even remote machines running on Amazon's EC2 infrastructure.

The `vagrant` command lets you manage your VMs (starting them, stopping them, destroying unneeded VMs, and so on), and you can log in to a VM simply by typing `vagrant ssh`. You can also share directories (such as your software's source code repository) between the host machine and the VM, and Vagrant can forward ports from the VM to the host machine, so you could access a web server running on the VM by accessing http://localhost/ on your local machine.

The main benefits of using Vagrant are as follows.

- It makes it easy to automate the setup of a development environment inside a VM, as you'll see shortly.
- Each VM is isolated from the host machine and other VMs, so you don't need to worry about version conflicts when you have many different projects set up on the same machine. If one project needs Python 2.6, Ruby 1.8.1, and PostgreSQL 9.1, while another needs Python 2.7, Ruby 2.0, and PostgreSQL 9.3, it can be tricky to set everything up on your development machine. But if each project lives in a separate VM, it can make life easier.

- The VMs are usually Linux machines, so if you're using Linux in production, you can exactly recreate the production environment.

If you want to get really fancy, Vagrant even supports multi-VM setups, so you could build the entire stack for your application (including web servers, DB servers, cache servers, Elasticsearch clusters, and what have you), exactly replicating the setup you have in production, but all running inside your development machine!

If you want to follow along with the rest of the chapter and you don't have Vagrant installed, head over to the Vagrant website (www.vagrantup.com/) and follow the installation instructions. It's pretty simple. Note that you'll also need a VM provider such as VirtualBox or VMware installed. For the remainder of the chapter I'll be using VirtualBox.

7.3.2 Setting up Vagrant for the UAD project

To add Vagrant support to the UAD, you first need to create a Vagrantfile. This is a file in the root folder of the repository named, unsurprisingly, `Vagrantfile`. It's a configuration file, written in a Ruby DSL, that tells Vagrant how to set up the VM for this project.

You can create a new Vagrantfile by running `vagrant init`. A minimal Vagrantfile is shown here:

```
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  config.vm.box = "ubuntu/trusty64"
end
```

Note that you need to specify what box to use for your VM. A box is a base image that Vagrant can use as a foundation for building a new VM. I'll be using 64-bit Ubuntu 14.04 (Trusty Tahr) as the OS for my virtual machine, so I set the box to `ubuntu/trusty64`. There are many other boxes available on the Vagrant website.

WATCH OUT FOR SPACES IN THE PATH The code shown in the remainder of this chapter won't work (and you'll get some confusing error messages) if you have spaces anywhere in the path of your working directory. If you want to follow along at home, make sure you don't get caught out by this.

Now you're ready to start your VM by typing `vagrant up`. Once it boots, you can log in by typing `vagrant ssh` and take a look around.

There's not much to see yet, but one thing to notice is that the folder containing the Vagrantfile has been automatically shared, so it's available as `/vagrant` inside the VM. This is a two-way share, so any changes you make in the VM will be reflected in real time on your host machine, and vice versa.

CODE ONLINE You can see the complete code for this chapter in the GitHub repo (https://github.com/cb372/ReengLegacySoft).

So far Vagrant isn't doing anything very useful, as we have only an empty Linux machine. The next step is to automate the installation and configuration of the UAD's dependencies.

7.3.3 Automatic provisioning using Ansible

The installation and configuration of everything needed to run a piece of software is known as *provisioning*. Vagrant supports a number of ways of provisioning, including Chef, Puppet, Docker, Ansible, and even plain old shell scripts.

For simple tasks, a bunch of shell scripts is often good enough. But they're difficult to compose and reuse, so if you want to do more complex provisioning or reuse parts of the provisioning script across multiple projects or environments, it's a good idea to use a more powerful tool. In this book I'll be using Ansible, but you can achieve much the same thing using Docker, Chef, Puppet, Salt, or whatever tool you're happiest with.

In this chapter we're going to write a few Ansible scripts to provision the UAD application, and in chapter 8 we'll reuse those scripts so we can perform exactly the same provisioning across all our environments, from the local development machine all the way through to production.

Before we can provision with Ansible, we need to install it on the host machine. See the installation docs on the Ansible website for details (http://docs.ansible.com/intro_installation.html). (I appreciate the irony of manually installing all this stuff so we can automate the installation of other stuff, but I promise this is the last thing you'll need to install manually. And after you've installed VirtualBox, Vagrant, and Ansible once, you can use them for all your projects.)

> **ANSIBLE ON WINDOWS** Ansible doesn't officially support running on Windows, but with a bit of work it's possible to get it running. The Azavea Labs blog has an excellent step-by-step guide on getting Vagrant and Ansible working on Windows: "Running Vagrant with Ansible Provisioning on Windows" (http://mng.bz/WM84).

Unlike other provisioning tools such as Chef or Puppet, Ansible is agentless. This means you don't need to install any Ansible agent on your Vagrant VM. Instead, whenever you run Ansible, it will execute commands on the VM remotely using SSH.

To tell Ansible what to install on your VM, you need to write a YAML file called a playbook, which we'll save as provisioning/playbook.yml. A minimal example is shown here.

```
---
- hosts: all
  tasks:
    - name: Print Hello world
      debug: msg="Hello world"
```

This tells Ansible two things. First, it should run the script on all hosts that it knows about. In our case, we only have a single VM, so this is fine for our purposes. Second, it should run a task that prints "Hello world".

THE YAML FORMAT Ansible files are all written in the YAML format. Indentation is used to represent the structure of your data, and you must use spaces (not tabs) for indentation.

You'll also need to add a couple of lines to your Vagrantfile to tell Vagrant to use Ansible for provisioning. Your Vagrantfile should now look like this.

```
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  config.vm.box = "ubuntu/trusty64"

  config.vm.provision "ansible" do |ansible|
    ansible.playbook = "provisioning/playbook.yml"
  end
end
```

Now if you run `vagrant provision`, you should see output something like the following.

```
PLAY [all] ********************************************************************

GATHERING FACTS ***************************************************************
ok: [default]

TASK: [Print Hello world] *****************************************************
ok: [default] => {
    "msg": "Hello world"
}

PLAY RECAP ********************************************************************
default                    : ok=2    changed=0    unreachable=0    failed=0
```

Now that you've got Ansible hooked up to Vagrant, you can use it to install the dependencies for the UAD. Recall that you need to do the following:

- Install Java
- Install Apache Ant
- Install Redis
- Install Resin 3.x
- Install and configure Apache ActiveMQ
- Download a license file and copy it to the Resin installation folder

We'll use the concept of Ansible roles, creating a separate role for each of these dependencies. This keeps each dependency cleanly separated, so we can later reuse them individually if we wish. Let's start with Java, as we need that before we can do much else.

OpenJDK can be installed using the `apt` package manager in Ubuntu, so our Java role will be quite simple. It will have just one task that installs the `opendjk-7-jdk` package.

Let's create a new file, provisioning/roles/java/tasks/main.yml (by convention this is where Ansible will look for the Java role's tasks), and write our task there:

```
---
- name: install OpenJDK 7 JDK
  apt: name=openjdk-7-jdk state=present
```

There are a couple of things to note, even in this very short file. First, `apt` is the name of a built-in Ansible module. There are loads of these, and it's worth becoming familiar with them so you don't accidentally reinvent the wheel when there's already a module that does what you want. You can see a list of them, with documentation and examples, on the Ansible website (http://docs.ansible.com/list_of_all_modules.html).

Second, you're not actually telling Ansible to install Java, but rather to ensure the Java package is present. Ansible is smart enough to check if the package is already installed before it tries to install it. This means that (well-written) Ansible playbooks are idempotent, so you can run them as many times as you like.

We need to tell the playbook to use our new Java role, so let's update the provisioning/playbook.yml file. It should now look like this.

```
---
- hosts: all
  sudo: yes
  roles:
    - java
```

Now if you run `vagrant provision` again, the output should look something like this.

```
PLAY [all] ********************************************************************
ATHERING FACTS ****************************************************************
ok: [default]

TASK: [java | install OpenJDK 7 JDK] ******************************************
changed: [default]

PLAY RECAP ********************************************************************
default                    : ok=2    changed=1    unreachable=0    failed=0
```

If you want to check that it worked, SSH into the VM and run `java -version`:

```
vagrant@vagrant-ubuntu-trusty-64:~$ java -version
java version "1.7.0_79"
OpenJDK Runtime Environment (IcedTea 2.5.5) (7u79-2.5.5-0ubuntu0.14.04.2)
OpenJDK 64-Bit Server VM (build 24.79-b02, mixed mode)
```

Cool! You just installed your first dependency using Vagrant and Ansible.

7.3.4 Adding more roles

Let's continue in the same vein, adding another role for each of the dependencies. Next in the list are Redis and Ant, but they're pretty much the same as Java (just installing a package using `apt`), so I'll gloss over them here. Remember, you can view the complete code for this chapter in the GitHub repo (https://github.com/cb372/ReengLegacySoft).

We'll try Resin next. The Resin role's tasks file is shown in the following listing. This file should be saved as provisioning/roles/resin/tasks/main.yml.

Listing 7.2 Ansible tasks to install Resin 3.x

```
---
- name: download Resin tarball
  get_url: >
    url=http://www.caucho.com/download/resin-3.1.14.tar.gz
    dest=/tmp/resin-3.1.14.tar.gz

- name: extract Resin tarball
  unarchive: >
    src=/tmp/resin-3.1.14.tar.gz
    dest=/usr/local
    copy=no

- name: change owner of Resin files
  file: >
    state=directory
    path=/usr/local/resin-3.1.14
    owner=vagrant
    group=vagrant
    recurse=yes

- name: create /usr/local/resin symlink
  file: >
    state=link
    src=/usr/local/resin-3.1.14
    path=/usr/local/resin

- name: set RESIN_3_HOME env var
  lineinfile: >
    state=present
    dest=/etc/profile.d/resin_3_home.sh
    line='export RESIN_3_HOME=/usr/local/resin'
    create=yes
```

This file is much longer than the previous one, but if you look at each task in turn, you'll see that it's not doing anything too complicated. The tasks, which will be run by Ansible in the order they're written, are doing the following:

1. Downloading a tarball from the Resin website
2. Extracting it under /usr/local
3. Changing its owner from `root` to the `vagrant` user
4. Creating a convenient symlink at /usr/local/resin
5. Setting up the `RESIN_3_HOME` environment variable that the UAD application requires

If you add the new Resin role to the main playbook file and run `vagrant provision` again, you should end up with Resin installed and ready to run.

The tasks for the next role, ActiveMQ, are similar to those for installing Resin (download a tarball, extract it, and create a symlink). The only task of note is the final one:

```
- name: customize ActiveMQ configuration
  copy: >
    src=activemq-custom-config.xml
    dest=/usr/local/activemq/conf/activemq.xml
    backup=yes
    owner=vagrant
    group=vagrant
```

This task uses Ansible's `copy` module, which copies a file from the host machine to the VM. You use this to overwrite ActiveMQ's configuration file with a customized one after the tarball has been extracted. This is a common technique, in which large files are downloaded from the internet onto the VM, but smaller files, such as configuration files, are stored in the repository and copied from the host machine.

The only remaining task is to download a license file for a proprietary XML parsing library from somewhere on the company's internal network, and store it in the Resin root directory. This task is quite specific to the UAD application and likely can't be reused anywhere, so let's create a role just for UAD-specific stuff and put it in there.

I'll leave the task definition as an exercise for the reader, in case you want to practice writing Ansible scripts. (Download any random text file from the internet to represent the hypothetical license file.) A solution is available in the GitHub repo.

7.3.5 Removing the dependency on an external database

This is going great so far. We've managed to automate almost the entire setup of the UAD development environment with just a few short YAML files, which should make the process a lot less painful for the next person who has to set this project up on their machine.

But there's one last issue that we haven't tackled. As things stand, the software depends on a shared PostgreSQL database in the test environment, so all new starters need to ask the ops team to create a DB user for them. If we could set up a PostgreSQL DB inside the VM and tell the software to use that one instead, it would solve the problem. It would also mean that each developer would have complete control over the content of their DB, without fear of somebody else tampering with their data. Let's give it a try!

We'll assume we have some credentials for the test environment and use those credentials to connect to the DB and take a dump of the schema:

```
$ pg_dump --username chris --host=testdb --dbname=uad --schema-only > schema.sql
```

Then we'll add some Ansible tasks to do the following: install PostgreSQL, create a DB user, create an empty DB, and initialize it using the schema.sql file we just generated. This is shown in the following listing.

Listing 7.3 Ansible tasks to create and initialize a PostgreSQL database

```
- name: install PostgreSQL
  apt: name={{item}} state=present
  with_items: [ 'postgresql', 'libpq-dev', 'python-psycopg2' ]

- name: create DB user
  sudo_user: postgres
  postgresql_user: >
    name=vagrant
    password=abc
    role_attr_flags=LOGIN

- name: create the DB
  sudo_user: postgres
  postgresql_db: >
    name=uad
    owner=vagrant

- name: count DB tables
  sudo_user: postgres
  command: >
    psql uad -t -A
    -c "SELECT count(1) FROM pg_catalog.pg_tables \
        WHERE schemaname='public'"
  register: table_count

- name: copy the DB schema file if it is needed
  copy: >
    src=schema.sql
    dest=/tmp/schema.sql
  when: table_count.stdout | int == 0

- name: load the DB schema if it is not already loaded
  sudo_user: vagrant
  command: psql uad -f /tmp/schema.sql
  when: table_count.stdout | int == 0
```

The psycopg2 library is needed to use the postgresql_* modules.

Uses the number of DB tables to decide if you have already loaded the schema

This task will not run if the DB schema already contains tables.

Note that this is a little more complicated than the Ansible tasks we've written so far, because we need to do a bit of trickery to achieve idempotency. This listing does some conditional processing so that you only load the DB schema if the number of tables in the DB is zero, meaning that you haven't already loaded it.

We've now automated the creation of a local PostgreSQL DB, so we've filled in the final piece of the automation puzzle. In the next section we'll look at how this automation effort pays off.

7.3.6 First day on the job—take two

Congratulations, and welcome to your new job at Fzzle, Inc. Once the HR guy has given you the tour of the office, he takes you to meet your new boss. Anna is the tech

lead of the User Services team. She shows you your desk and fills you in on the details of the job.

Your first task is to add a new feature to an application called the User Activity Dashboard. You clone the Git repository and take a look at the README file to see how to get it running locally.

The README explains that you can set up a development environment using Vagrant and Ansible. As standard elements of the company's recommended tool chain, these tools are preinstalled on your development machine. You kick off the `vagrant up` command, which will build and provision your VM. It'll take a few minutes to complete, so you wander off to figure out how the coffee machine works ...

By the time you get back, the provisioning is complete and you get the application running with little fuss. By lunchtime you start work on implementing the new feature. By the end of the day you've completed the implementation and made your first pull request, and you've also made a note of a couple of places you'd like to refactor tomorrow. Not a bad first day on the job!

7.4 Summary

- The README is the most important file in the repository.
- Making it easy for a new developer to get started will make people more likely to contribute.
- Vagrant and Ansible are useful tools for automating the provisioning of a development environment for an application.
- Where possible, you should remove dependencies on shared developer DBs and other external resources. The Vagrant VM should contain everything the application needs, so that it's all under the developer's control.

SOFTWARE ENGINEERING

Re-Engineering Legacy Software

Chris Birchall

As a developer, you may inherit projects built on existing codebases with design patterns, usage assumptions, infrastructure, and tooling from another time and another team. Fortunately, there are ways to breathe new life into legacy projects so you can maintain, improve, and scale them without fighting their limitations.

Re-Engineering Legacy Software is an experience-driven guide to revitalizing inherited projects. It covers refactoring, quality metrics, toolchain and workflow, continuous integration, infrastructure automation, and organizational culture. You'll learn techniques for introducing dependency injection for code modularity, quantitatively measuring quality, and automating infrastructure. You'll also develop practical processes for deciding whether to rewrite or refactor, organizing teams, and convincing management that quality matters. Core topics include deciphering and modularizing awkward code structures, integrating and automating tests, replacing outdated build systems, and using tools like Vagrant and Ansible for infrastructure automation.

What's Inside

- Refactoring legacy codebases
- Continuous inspection and integration
- Automating legacy infrastructure
- New tests for old code
- Modularizing monolithic projects

This book is written for developers and team leads comfortable with an OO language like Java or C#.

Chris Birchall is a senior developer at the *Guardian* in London, working on the back-end services that power the website.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/re-engineering-legacy-software

"A comprehensive guide to turning legacy software into modern, up-to-date projects!"
—Jean-François Morin
Université Laval

"I learn something new every time I read it."
—Lorrie MacKinnon
Treasury Board Secretariat
Province of Ontario

"The book I wanted to read years ago."
—Ferdinando Santacroce, 7Pixel

"A very practical guide to one of the most difficult aspects of software development."
—William E. Wheeler
West Corporation

ISBN-13: 978-1-61729-250-7
ISBN 10: 1-61729-250-8
56499
9 781617 292507

MANNING $64.99 / Can $74.99 [INCLUDING eBOOK]