Using Shell as a Deployment Tool

Ivan "Rambius" Ivanov

New York City BSD User Group

Feb, 2019

What Will I Talk About

Moving from ansible to pure shell scripting to manage our application's QA environment - how and why.

The Application

A financial application / trading platform consisting of a

- databases
- Linux executables
- Windows executables

The Environment

A QA environment for our application that hosts the manual / integration testing of the application.

The Starting Point

A set of ansible playbooks that deploy the application to the environment.

The Deployment Procedure

Not so complicated steps except the database parts:

- Setup the database
- Download binaries from the build servers and copy them to the boxes
- Apply the database changes' corresponding to those binaries
- Start the binaries
- Run the manual or integration tests
- Shutdown and collect the logs

Setup the Database Step

- Delete existing data from the database
- Import latest production data
- Run a set of predefined sql scripts to prepare the production data for testing

Download the Binaries Step

- Issue HTTP requests to download artifacts from the build server.
- Download from a specific branch or from a default branch.
- Download a specific version or the latest one.
- In case of the latest version use a specially crafted URL.
- Copy them to the boxes.

Apply the DB Changes for the New Binaries

- The DB changes are stored as sql scripts in VCS
- Changes in those sql scripts triggers builds so that the binaries and the sql scripts stay together
- Downloaded from the build server as well and then applied

How Ansible Works

- Executes tasks on remote hosts from a control host
  - SSH
  - WinRM
- The remote hosts are collected in an inventory
- Plays specifies what commands are run on what hosts
- The plays are collected in playbooks
- One central host orchestrates the commands' executions on the remote hosts

Ansible Implementation - Database Setup

Ansible has native modules for managing and querying various databases, but not for Oracle. Call Oracle utilities as shell commands.

Ansible - Run an SQL Script

```
- name: run sql script
  shell: "sqlplus {{ dbuser }}/{{  dbpass }}@{{ sid }}
          @file.sql"
  args:
    chdir: sqldir
```

Ansible - Import / Export DB

To import data into a database:

```
- name: import dump
  shell: impdp {{ dbuser }}/{{  dbpass }}@{{ sid }} ...
```

To export a database:

```
- name: export db
  shell: expdp {{ dbuser }}/{{  dbpass }}@{{ sid }} ...
```

Ansible - Fetch and Extract Binaries

The build server exports the build artifacts over HTTP. To download them:

```
- name: download archive
  get_url:
    url: http://buildserver/path/to/file.tgz
    dest: file.tgz
```

To extract them:

```
- name: extract archive
  unarchive:
    src: file.tgz
    dest: /path/to/extracted/files
```

Ansible - Sync Binaries

Ansible provides a wrapper around `rsync` to copy files to remote hosts.

```
- name: copy binaries
  synchronize:
    src: /path/to/extracted/files
    dest: /deploy
```

Ansible - Start / Stop Binaries

We start the application via shell scripts. The shell scripts knows what binaries to start on each box:

```
- name: create logs dir
  file: /deploy/logs state=directory
- name: start
  shell: start_app.sh
  args:
    chdir: /deploy

- name: stop
  shell: stop_app.sh
  args:
    chdir: /deploy
```

Ansible - Copy and Clean Logs

After the application is shut down, gather the logs. Unfortunately, ansible `fetch` task does not work with directories, need to use `scp` or `rsync` commands:

```
- name: copy logs
  fetch:
    dest: logs
    src: /deploy/logs <- does not work with dirs

- name: copy logs
  shell: "scp -r {{ ansible_hostname }}:/deploy/logs logs"
```

Ansible - Completed

At this point the deployment procedure implemented in Ansible was working more or less in a manageable and predictable way. There were some rough edges though.

Trivial Issues

- Yet another tool to learn, maintain and keep up-to-date
- Yet another syntax to learn - YAML
  - Chasing white-space issues in YAML is not fun

Shell Commands Anyway

Due to missing functionality, like an Oracle module, or incomplete one, like `fetch` task, we still need to execute pure shell commands in the plays with either `shell` or `command` tasks:

```
- name: executes some script
  shell: somescript.sh
- name: executes some command
  command: somecmd.sh
```

`command` task does not process the command through shell, so pipes and shell substitutions are not available.

Too Verbose when Processing Output

```
- name: execute some script
  shell: somescript.sh
  register: result
- name: if output contains
  sometask: ...
  when: "'success' in {{ output.stdout }}"
- name: if output does not contain
  othertask: ...
  when: "'sucess' not in {{ output.stdout }}"
```

Compare with pipes and `grep` and `if`.

Too Verbose when Processing Exit Codes

```
- name: execute some script
  shell: somescript.sh
  register: result
  ignore_errors: True
- name: if rc is not 0
  sometask:
  when: result.rc != 0
```

Compare with `if command`.

Defaults and Options - I

The application relies on external services, located by configurable values in the database. A service may have several variants:

- simulator
- real
- testing

Default values in the playbooks can be overridden from the command line:

```
ansible-playbook site.yml --extra-vars "srv1_host=...\
  srv1_port=...\
  srv2_host=... srv2_port\
  srv3_url=..."
```

Too much options' names to remember and type :(

Defaults and Options - II

Use YAML dictionaries to encapsulate services' values:

```
vars:
  srv1_sim:
    host: ...
    port: ...
  srv1_real:
    host: ...
    port: ...
  srv3_real:
    url: ...
  srv3_test:
    url: ...
```

Defaults and Options - III

Call `ansible-playbook` with logical names for the services

```
ansible-playbook site.yml --extra-vars "srv1=srv1_sim\
  srv3=srv3_test"
```

Compare all that machinery with `getopt` in shell.

Moving to Shell Scripting

We decide to move to shell scripting, because we felt the implementation will be simpler.

Shell - The Basic Structure

The ansible implementation albeit cumbersome helped us split the procedure into steps that translated into shell functions in the main script `driver.sh`

```
dbimport() {
   ...
}
download_binaries() {
   ...
}
deploy_binaries() {
   ...
}
```

Shell - Functions as Subcommands

To call those functions:

```
subcmd=$1
shift
$subcmd $@
ec=$?
exit $ec
```

For example

```
$ ./driver.sh dbimport opts args
```

Each function / subcommand parses its own options and arguments usually with `getopt`.

Shell - "Private" Functions

What if some functions should not be called as subsommands? Prefix the function name with __ and check for that:

```
__errmsg() {
  echo $@ >&2
}

subcmd=$1
shift
if expr "$subcmd" : "^__" > /dev/null ; then
  __errmsg "$subcmd is private; cannot call it"
  exit 1
fi

$ ./driver.sh __errmsg test
__errmsg is private; cannot call it
```

Shell - Non-existing Subcommands

Calling a non-existing function throws 127 status

```
$subcmd $@
ec=$?
if [ $ec = 127 ]; then
  __errmsg "Subcommand $subcmd does not exit"
  exit $ec
fi
exit $ec
```

Shell - Exit Codes

Other scripts can call `driver.sh` and should be able to check for errors. We exit each subcommand with different statuses on different errors. Example from a script called by a cron job:

```
if ./driver.sh dbimport opts args
then
  proceed()
else
  cat logs/import.log | mail -s "DB Import Failed" \
     all@team.com
  exit 1
fi
```

Shell - Error Handling 1

A basic way to exit on error is `set -e`. It exits when an untested command fails. However, it does not allow a corrective action.

```
echo "before false"
false
echo "after false"
```

```
set -e
echo "before false"
false
echo "after false"
```

Shell - Error Handling 2

Testing a command with `if`:

```
if ./driver.sh dbimport opts args
then
  process()
else
  correct_or_exit()
fi
```

or simply

```
if ! ./driver.sh dbimport opts args
then
  correct_or_exit()
fi
```

Shell - Error Handling 3

```
# Buy Ike a beer
yell() { echo "$0: $*" >&2; }
die() { yell "$*"; exit 111; }
try() { "$@" || die "cannot $*"; }
try ./driver.sh dbimport opts args
```

See NYCBUG presentation from 2016-02-03 for more information.

Simplifying DB Operations - 1

In the Ansible implementation we picked a random host where we ran all DB operations:

```
- hosts: dbhost
  tasks:
  - name: copy sql files
    copy: src={{ item }} dest=~/sqldir
    with_fileglob:
    - files/*.sql
  - name: run file1.sql
    shell: "sqlplus {{ dbuser }}/{{  dbpass }}@{{ sid }}  \
          @file1.sql"
```

No reason to first copy the files and then execute them.

Simplifying DB Operations - 2

If I have to implement that in Ansible now, I will use either `local_action` or

```
- hosts: localhost
  tasks:
  - name: run file1.sql
    shell: "sqlplus {{ dbuser }}/{{  dbpass }}@{{ sid }}  \
          @file1.sql"
  - name: run file2.sql
    shell: "sqlplus {{ dbuser }}/{{  dbpass }}@{{ sid }}  \
          @file2.sql"
```

DB Operations in Shell - 1

The central command for running sql files is:

```
run_sqlplus() {
  if [ $# -ne 4 ]; then
    __errmsg "Illegal arguments"
    exit 1
  fi
  user=$1
  pass=$2
  sid=$3
  file=$4
  log="$logsdir/‘basename $file‘.log"
  sqlplus $user/$pass@$sid @$file > $log
}
```

DB Operations in Shell - 2

Call `run_sqlplus` as

```
run_sqlplus dbuser dbpass dbsid sqldir/file.sql
```

Again no one wants to type so many arguments.

DB Operations in Shell - 3

As we have a limited number of databases we can hardcode their info in `driver.sh`:

```
db1_user="user1"
db1_pass="pass1"
db1_sid="sid1"

db2_user="user2"
db2_pass="pass2"
db2_sid="sid2"
```

Disclaimer

We would never hardcode production databases' passwords in a shell script, but these are test databases containing no valuable information. Protecting their credentials will require more effort than it is worth.

DB Operations in Shell - 4

Now we pass only the prefix of the database variables' names:

```
run_sqlplus_prefix() {
  if [ $# -ne 2 ]; then
    __errmsg "Illegal arguments"
    exit 1
  fi
  prefix=$1
  file=$2
  eval "user=\${${prefix}_user}"
  eval "pass=\${${prefix}_pass}"
  eval "sid=\${${prefix}_sid}"
  run_sqlplus $user $pass $sid $file
}
```

DB Operations in Shell - 5

Call `run_sqlplus_prefix` as

```
run_sqlplus_prefix db1 file.sql
run_sqlplus_prefix db2 other.sql
```

DB Import / Export Utilties in Shell

We have similar function for `expdp` and `impdp` and Oracle utilities that dump a database's content and import a dump in a database:

```
run_impdp user pass sid dumpname
run_impdp_prefix dbprefix dumpname

run_expdp user pass sid dumpname
run_expdp_prefix dbprefix dumpname
```

Using the DB Functions

Example with importing a production datbase dump:

```
dbimport() {
  prefix1=$1
  prefix2=$2
  run_sqlplus_prefix $prefix1 purge_db.sql
  run_sqlplus_prefix $prefix2 purge_db.sql

  run_impdp_prefix $prefix1 full_data.dmp
  run_impdp_prefix $prefix2 ddl_only.dmp

  run_sqlplus_prefix $prefix1 prepare.sql
  run_sqlplus_prefix $prefix1 preparemore.sql
}
```

Deploying the Binaries

After the database is ready we are ready to deploy the binaries to the boxes.

Downloading the Binaries - Specific Version

To download a specific version use a simple call to `curl`:

```
download_binaries() {
  version=$1
  baseurl=http://buildserver/project
  url=$baseurl/app-$version.tar.gz
  curl -sS $url -o app-$version.tar.gz
}
download_binaries 1.2.3
```

`curl` options:

- -s - silent mode
- -S - when in silent mode show an error message if failure

Download the Binaries - the Latest Version

If we want to deploy the latest version `download_binaries` gets more complicated

```
download_binaries() {
  if [ $# -eq 0]; then
    version=‘resolve_version‘
  else
    version=$1
  fi
  baseurl=http://buildserver/project
  url=$baseurl/app-$version.tar.gz
  curl -sS $url -o app-$version.tar.gz
}
```

Resolve the latest version

`resolve_version` queries a special url and parse the headers to find the real version:

```
resolve_version() {
  baseurl=http://buildserver/project
  url=$baseurl/app-{build.number}.tar.gz
  echo `curl -sS -D - -g $url \
    | sed -n "s/.*app-\(.*\).tar.gz/\1/p"`
}
```

`curl` options:

- -D - dump the headers
- - - stdin
- -g - the urls can contain { and }

Inventory File - The Good Stuff From Ansible

Ansible keeps the list of the hosts it controls in an inventory file usually one host per line. The inventory can also contain a great amount of configuration as well and it can collect hosts in named groups.
Our inventory is just one host per line:

```
host1
host2
...
hostN
```

Copy the Binaries

"Load" the inventory and use `scp`

```
hosts=‘cat $hostfile‘
for host in $hosts
do
  scp -q ./app-$version/bin/* $host:/deploy/bin
  echo "Deployed to $host"
done
```

Parallelism

Ansible executes a task on the remote hosts in parallel. Some degree of parallelism can be achieved in shell as well:

```
for host in $hosts
do
  scp -q ./app-$version/bin/* $host:/deploy/bin && \
    echo "Deployed to $host" &
done
wait
```

Starting the Application

We start the application by calling a script on each box:

```
for host in $hosts
do
  ssh -q $host /deploy/bin/start_app.sh
done
```

Running Additional Commands

Often we need to run several commands on a single host:

```
ssh hostX cmd1
ssh hostX cmd2
...
```

It makes sense to reuse the ssh connection once created.

Reusing an SSH Connection - 1

To share an ssh connection to a host, define the following in `/.ssh/config`

```
Host hostX
ControlMaster auto
ControlPath ~/.ssh/sockets/%r@%h-%p
ControlPersist 600
```

Reusing an SSH Connection - 2

Or use `ssh` command-line options:

```
# create the sockect
ssh -S ~/.ssh/sockets/hostX -Mn hostX
# use the socket
ssh -S ~/.ssh/sockets/hostX hostX cmd1
ssh -S ~/.ssh/sockets/hostX hostX cmd1
```

Conclusion

Is Ansible a good tool? Yes!
We still use Ansible to handle setup of Windows boxes and for procedure that are more linear with less branching and conditionals.

Questions

Questions?