21 Jan 2014

Continuous Integration and Deployment with Jenkins and Docker (Part II)

In the first post I discussed the problems we were trying to solve, the tools we are using, and the installation of Jenkins. In this post I'll go into a lot more detail about the specific configuration of Jenkins and Docker.

Jenkins Configuration (Master)

The "master" Jenkins server builds the server

component of our system (the one written in Java). It performs the following tasks:

1. Monitor a single GIT repository and one or more branches of that repository for changes
2. When a change occurs, build the branch and run all JUnit tests (using ANT for both)
3. If there is a currently running Docker instance for the branch, export the Mongo database of that instance
4. Stop currently running Docker instance (if any)
5. Start Docker instance for new build
6. Export exposed ports of the running Docker instance to a property file that will be exported later to the slave Jenkins server running on Windows
7. Create a "buildinfo.html" file containing the settings of the running Docker instance so that users can access the webpage to configure the running server instance
8. Create a zip file containing the server build and post it to an internal web server for download
9. Trigger build of client on the Jenkins slave

I won't cover the first 2 steps since they're pretty straighforward. For the third step we use the following bash script to find the exposed MongoDB port of the

running Docker instance for the branch being built and export the databse to the "dbrestore" directory. This directory is loaded by the application start script described in the Docker configuration section.

```
#!/bin/bash

service=$JOB_NAME
service_port=9000
branch=$(echo $GIT_BRANCH | cut -d/ -f 2)

existing=$(docker ps | grep $branch | grep
-o "^[0-9a-z]*")
mongo_port=$(docker port $existing 27017 |
grep -o ":[0-9]*" | cut -d : -f 2)
if [ -n "$mongo_port" ]; then
  mongodump -h 127.0.0.1:$mongo_port -o
dist/dbrestore
fi
```

For steps 4 through 6 we use the following bash script:

```
#!/bin/bash

service=$JOB_NAME
service_port=9000
```

```
branch=$(echo $GIT_BRANCH | cut -d/ -f 2)

existing=$(docker ps | grep $branch | grep
-o "^[0-9a-z]*")
if [ ! -z "$existing" ]; then docker stop
$existing; fi

docker build -t $service:$branch $WORKSPACE

container_id=$(docker run -d -p
$service_port -p 27017 -p 5672
$service:$branch)
container_port=$(docker port $container_id
9000 | grep -o ":[0-9]*" | cut -d : -f 2)
mongo_port=$(docker port $container_id
27017| grep -o ":[0-9]*" | cut -d : -f 2)
rabbit_port=$(docker port $container_id 5672
| grep -o ":[0-9]*" | cut -d : -f 2)

echo -ne
"RABBIT_PORT=$rabbit_port\nMONGO_PORT=$mongo_po
\nCONTAINER_PORT=$container_port
\nDOCKER_ID=$container_id" >
container_env.prop

echo "Mongo Port: $mongo_port"
echo "Rabbit Port: $rabbit_port"
```

```
echo "App running on
http://localhost:$container_port"
```

This script finds and stops the running Docker instance for the branch being built if one exists. Then it builds and tags a new Docker instance for the current branch. After that it exports the ports exposed by the Docker instance (the ports exposed using the EXPOSE command in the Dockerfile explained later) to a property file. These variables are used later in the Jenkins build and are passed to the slave Jenkins instance so that the client can be correctly configured to connect to the server instance.

We're using the EnvInject Jenkins plugin to inject the exported environment variables (in container_env.prp) into the local environment and the Parameterized Trigger Plugin to provide these environment variables to the slave Jenkins slave.

Steps 7 and 8 are performed by the following bash script:

```
#!/bin/bash

branch=$(echo $GIT_BRANCH | cut -d/ -f 2)
```

```
zipname=$JOB_NAME-$branch
builddate=$(date -u)
echo -e "---\n{port: $CONTAINER_PORT,
rabbitport: $RABBIT_PORT, branch: $branch,
zipname: $zipname, builddate:
$builddate}\n---" | mustache -
buildinfo.mustache > buildinfo.html

echo "Zip file name: $zipname.zip"
zip -rq $zipname.zip dist/
```

We're using a mustache template to create a build information HTML file that developers and non-developers can use to access information about the currently running instance as well as download a zip file of the build.

After the server is built, a "parameterized build" (using the Parameterized Trigger Plugin) is triggered and the environment variables stored in "container_env.prop" are passed to the slave.

Docker Configuration

A Docker instance is configured using a "Dockerfile" which sets up the VM image that will run your software.

Fortunately, Docker's documentation is pretty great so this isn't too difficult. The Dockerfile for the project is stored in the repository so different branches can define a different Docker configurations.

I'll show our Dockerfile first and then explain the key parts of it. I won't explain every line; for that you should reference the Docker documentation.

```
FROM    ubuntu:precise

RUN sed -i.bak 's/main$/main universe/'
/etc/apt/sources.list
RUN apt-get update

RUN apt-get -qy install wget supervisor

# MongoDB APT repository
RUN apt-key adv --keyserver
hkp://keyserver.ubuntu.com:80 --recv
7F0CEB10
RUN echo 'deb http://downloads-
distro.mongodb.org/repo/ubuntu-upstart dist
10gen' | tee /etc/apt/sources.list.d
/10gen.list
```

```
# Erlang APT repository (required by
RabbitMQ)
RUN wget -qO- http://packages.erlang-
solutions.com/debian/erlang_solutions.asc |
apt-key add -
RUN echo 'deb http://packages.erlang-
solutions.com/debian precise contrib' | tee
/etc/apt/sources.list.d/erlang.list

# Hack for initctl not being available in
Ubuntu
RUN dpkg-divert --local --rename --add
/sbin/initctl
RUN ln -s /bin/true /sbin/initctl

RUN apt-get -y update
RUN apt-get install -y mongodb-10gen=2.0.7

RUN apt-get install -y erlang-nox
RUN apt-get install -y logrotate
RUN wget -q http://www.rabbitmq.com/releases
/rabbitmq-server/v2.8.6/rabbitmq-
server_2.8.6-1_all.deb
RUN dpkg -i rabbitmq-server_2.8.6-1_all.deb
RUN apt-get -f install

# Install Java and set environment variables
```

```
RUN wget -q http://192.168.1.5/jre-7u45-
linux-x64.gz
RUN mkdir /etc/java
RUN tar -xvf jre-7u45-linux-x64.gz -C
/etc/java
ENV JAVA_HOME /etc/java/jre1.7.0_45
ENV PATH $JAVA_HOME/bin:$PATH

# Make the default MongoDB database dir
RUN mkdir -p /data/db

ADD ./dist /var/apps/server
ADD supervisord.conf /etc/supervisor/conf.d
/supervisord.conf

EXPOSE 9000 27017
CMD ["/usr/bin/supervisord"]
```

The majority of this file is concerned with installing the dependencies of our server component: Java 1.7, RabbitMQ, and MongoDB. We also install "supervisor" which is not required by our application but is used to start it and all of its dependencies when the Docker image is started.
After installing dependencies the server binaries that were built by Jenkins are copied over (the ADD command), ports required by the client application are

exposed (the EXPOSE command) and finally the supervisord application is run which takes care of starting all dependencies and the server application.

Java was kind of a pain to install using apt-get so we got around this by installing NGINX on the Linux CI server and locally hosting the JRE distribution required by our software.

We use Supervisor to start MongoDB, RabbitMQ, and our application after the Docker instance is started. Here's our configuration file:

```
[supervisord]
nodaemon=true

[program:mongodb]
command=/usr/bin/mongod

[program:rabbitmq]
command=/usr/sbin/rabbitmq-server

[program:server]
command=/var/apps/server/start-populatedb.sh
-f
directory=/var/apps/server
```

The "start-populatedb.sh" is a wrapper around our standard application start script. Its purpose is to restore a previous version of the Mongo database in between runs of the same branch. For example, if you enter a bunch of data into the application for the "develop" branch and then someone checks code into the "develop" branch, Jenkins would rebuild and redeploy this branch which would cause all of the data that you entered to be lost. All the "start-populatedb.sh" script does is use the mongorestore command to load the database image that was exported as part of the Jenkins build process (see previous section).

```
mongorestore ./dbrestore
/bin/sh ./start.sh
```

Jenkins Configuration (Slave)

The slave Jenkins instance runs on a Windows 7 machine. It needs to monitor one or more git branches for changes but can also be triggered by a build of the master Jenkins node. This node has to perform the following tasks:

1. Monitor one or more GIT branches for changes.
2. When a change occurs, or the master triggers a

build, build the branch using MSBuild
3. If the environment variables RABBIT*PORT and CONTAINER*PORT are set, export their values to a prp file. These variables will be set if the build was triggered by the master, otherwise they will not be set.
4. Load the prp file containing the RABBIT*PORT and CONTAINER*PORT variables into the environment. If the build was triggered by a change to the GIT repository (not by the master node) then the environment will not have these values so we have to load them from a previous build that was triggered by the master node.
5. Create a link on the desktop to the built application. Pass the RABBIT*PORT and CONTAINER*PORT environment variables as arguments to the application so that it connects to the right server.
6. Create a zip file containing the built application and publish it to the master Jenkins node so that users can easily download it.

The complicated part about building the client application was ensuring that it connects to the right running Docker instance containing the server application. The master Jenkins node will inject environment variables containing the ports for Rabbit and the REST interface when it triggers a build of the

client, but a build can also be triggered if the client GIT repository changes. In that case we won't have environment variables from the server so we save them to a PRP file when the server triggers a build and we load this PRP file when the client is built. We're assuming that the server was built at least once before the client is built (i.e. the server is built once but the client may be re-built many times).

The batch file that saves the RABBIT*PORT and CONTAINER*PORT environment variable to a PRP file if they are set in the environment looks like this:

```
IF NOT "%RABBIT_PORT%"=="" (
 IF NOT "%CONTAINER_PORT%"==""  (
   echo RABBIT_PORT=%RABBIT_PORT% >
saved_master_settings.properties
   echo CONTAINER_PORT=%CONTAINER_PORT% >>
saved_master_settings.properties
 )
)
```

This is the batch file that creates a shortcut on the desktop to the newly built client application and passes the RABBIT And CONTAINER ports as command line arguments to the application:

```
set
SCRIPT="%TEMP%\%RANDOM%-%RANDOM%-%RANDOM%-%RAND

echo Set oWS =
WScript.CreateObject("WScript.Shell") >>
%SCRIPT%
echo sLinkFile = "C:\Users\Public\Desktop
\Client-development.lnk" >> %SCRIPT%
echo Set oLink =
oWS.CreateShortcut(sLinkFile) >> %SCRIPT%
echo oLink.TargetPath = "%WORKSPACE%\client
\bin\Debug\client.exe" >> %SCRIPT%
echo oLink.Arguments = "192.168.1.5
%CONTAINER_PORT% 192.168.1.5 %RABBIT_PORT%"
>> %SCRIPT%%
echo oLink.Save >> %SCRIPT%

cscript /nologo %SCRIPT%
del %SCRIPT%
```

And finally, this is the batch file that copies a zip file containing the client build to the master Jenkins server for download.

```
set FOLDERTOZIP=%WORKSPACE%\client
\bin\Debug
```

```
for /f "tokens=1,2 delims=/ " %%a in
("%GIT_BRANCH%") do set repo=%%a&set
branchname=%%b
set
ZIPFILE=%WORKSPACE%\%JOB_NAME%-%branchname%.zip
set
SCRIPT="%TEMP%\%RANDOM%-%RANDOM%-%RANDOM%-%RAND

echo InputFolder = "%FOLDERTOZIP%" >
%SCRIPT%
echo ZipFile = "%ZIPFILE%" >> %SCRIPT%

echo
CreateObject("Scripting.FileSystemObject").Crea
True).Write "PK" ^& Chr(5) ^& Chr(6) ^&
String(18, vbNullChar) >> %SCRIPT%
echo Set objShell =
CreateObject("Shell.Application") >>
%SCRIPT%
echo Set source =
objShell.NameSpace(InputFolder).Items >>
%SCRIPT%
echo
objShell.NameSpace(ZipFile).CopyHere(source)
>> %SCRIPT%
echo wScript.Sleep 2000 >> %SCRIPT%
```

```
cscript /nologo %SCRIPT%

rem copy to server
pscp -i C:\Users\Jenkins
\.ssh\putty_id_rsa.ppk %ZIPFILE%
jenkins@192.168.1.5:/usr/share/nginx
/www/builds/
```

Conclusion

Overall I was pretty impressed with how easy it was to use Jenkins and Docker and to get them working together. I had to perform a lot more manual configuration of Jenkins (i.e. shell scripts and batch files) then I thought I would need to going into this, but that also allowed us to get Jenkins to do exactly what we needed. I'm not sure how we would have satisified the goal of running multiple, isolated, branches of the server on the same machine without the use of Docker.

I glossed over, or just skipped completely, a lot of small implementation details but hopefully I covered most of the important stuff. Good luck!

Adam Stull

CTO and Software Engineer at a small software

Share this post

startup creating software for RPAs (drones).
@inpursuit on twitter

Ghostery hat Kommentare von Disqus blockiert.

comments powered by Disqus