accusoft®

Using Gradle for Single Click Deployment

Isaac Yousuf
Software Engineer III

As a software engineer, nothing demotivates me more than having to test a web application remotely. Just imagining having to stop the application, upload the new code, and rerun the app tires me out. Fortunately, this process can be easily automated with build tools. Here I will share how I've overcome this issue by using one of my favorite tools: Gradle.

We will be deploying a simple "Hello World" web app. The entire source used here can be found on GitHub.

Requirements:

- Gradle
- A remote Linux machine (we will be using a Docker container)
  - Java 8+ installed
  - SSH access
- Basic command line knowledge

Project folder structure:

```
- webapp/
  - src/main/java/server/Main.java
  - deploy/
    - Dockerfile
    - setup_webserver.sh
  - build.gradle
  - gradle.properties
```

Step 1: Setting up our remote machine:

Note: If you already have a Linux server that you can SSH into, then skip to Step 2.

Let's begin by creating a Docker image using a Dockerfile. Following the folder structure shown previously, save this to webapp/deploy/Dockerfile:

Dockerfile:

```
FROM ubuntu:16.04

RUN apt-get update
RUN apt-get install -y openssh-server
RUN apt-get install -y unzip
RUN apt-get install -y default-jre

RUN mkdir /var/run/sshd
RUN echo 'root:password' | chpasswd

RUN useradd webadmin
RUN echo "webadmin:password" | chpasswd

RUN sed -i 's/PermitRootLogin prohibit-password/PermitRootLogin yes/' /etc/ssh/sshd_
config

# SSH login fix. Otherwise user is kicked off after login
RUN sed 's@session\s*required\s*pam_loginuid.so@session optional pam_loginuid.so@g' -i /
etc/pam.d/sshd

ENV NOTVISIBLE "in users profile"
RUN echo "export VISIBLE=now" >> /etc/profile

RUN mkdir /home/webadmin/
RUN chown -R webadmin:webadmin /home/webadmin/

EXPOSE 22 80
CMD ["/usr/sbin/sshd", "-D"]
```

To execute:

1. Open command line in webapp/deploy/
2. Execute: `docker build -t ssh_ubuntu ./`
3. Then execute: `docker run -t -d -p 8887:22 -p 8888:8888 --name webserver ssh_ubuntu`

The first command will create a Linux Docker image with all the requirements and a user called webadmin with password "password".

The second command will create a Docker container with the Linux image. Note: It will map the SSH port to 8887.

Step 2: Include Java project source code

Gradle requires a specific project folder structure. Following this folder structure, save this to webapp/src/main/java/server/Main.java:

```
package server;

import static spark.Spark.get;
import static spark.Spark.port;

public class Main {

    public static void main(String[] args) {
        port(8888);

        get("/", (req, res) -> {
            return "hello world!";
        });
    }

}
```

We are utilizing a Java library called Spark to create a very basic http server that responds with "hello world!" to GET requests to http://ip-address:8888.

Step 3: Gradle project files

There are two Gradle files required for this example: build.gradle and gradle.properties.

Step 3a: build.gradle

First, save the following to webapp/build.gradle:

```
plugins {
    id 'java'
    id 'application'
    id "org.hidetake.ssh" version "2.9.0"
}

sourceCompatibility = '1.8'
mainClassName = 'server.Main'

repositories {
    mavenCentral()
}

dependencies {
    compile group: 'com.sparkjava', name: 'spark-core', version: '2.7.2'
}

remotes {
    webServer {
        host = project.property('ssh.webServer.host')
        user = project.property('ssh.webServer.user')
        password = project.property('ssh.webServer.pass')
        port = project.property('ssh.webServer.port') as Integer
    }
}

applicationName = 'webserver'
distZip { archiveName "webserver.zip" }
task deploy(dependsOn: distZip) {
    doLast {
        ssh.settings {
            knownHosts = allowAnyHosts
            logging = 'stdout'
        }
        ssh.run {
            session(remotes.webServer) {
                put from: new File(buildDir, '/distributions/webserver.zip'),
                into: './'

                put from: new File(rootDir, 'deploy/setup_webserver.sh'),
                into: './'

                execute('pwd')
                execute('chmod +x setup_webserver.sh')
                execute('./setup_webserver.sh')
            }
        }
    }
}
```

This is the main gradle file, and contains various important features. For example, Java version, dependencies, library repositories, gradle plugins, and build tasks.

As you can see, this particular build.gradle has a “deploy” task, and “remotes” object. The remotes object contains the different remote servers for the SSH plugin. We will be using a gradle.properties file to store the remote server settings.

Step 3b: gradle.properties

Second, save the following to webapp/gradle.properties:

```
ssh.webServer.host=localhost
ssh.webServer.user=webadmin
ssh.webServer.pass=password
ssh.webServer.port=8887
```

Note: these are the settings for the Docker container created in step 1. If you skipped, and have your own remote Linux server, ensure that these properties match your information for SSH access.

At this point we should be able to build the web application by simply executing: gradle build. Once Gradle reports “BUILD SUCCESSFUL,” we should see a new directory called webapp/build. This directory contains the ready to deploy web application.

Step 4: Set up script

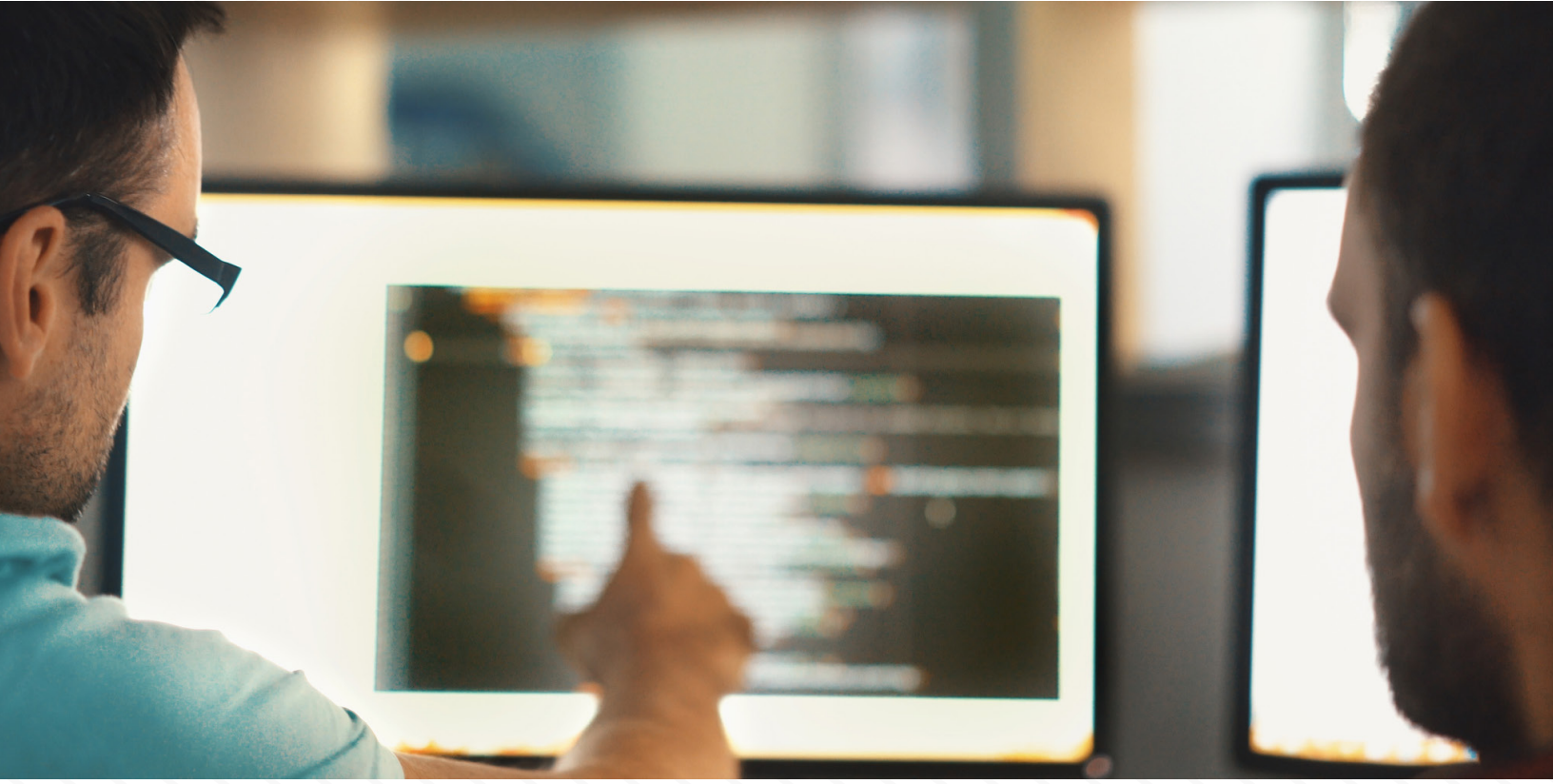
You may have noticed that in the webapp/build.gradle, the deploy task copies a file called setup_webserver.sh to our remote Linux server. This script simply handles starting/stopping our application for us automatically.

Save the following to webapp/deploy/setup_webserver.sh:

```
#!/bin/bash
# ~/setup.sh
#

set -e

serverDir="${HOME}/webserver"
serverZip="${HOME}/webserver.zip"
pidFile="${HOME}/webserver.pid"

function start_server() {
   echo "Starting webserver..."
   nohup $serverDir/bin/webserver > /dev/null 2>&1 & echo $! > $pidFile
   sleep 5
   echo "PID is $(cat $pidFile)"
}

function stop_server() {
   set +e
   echo "Stopping webserver (PID = $(cat $pidFile))"
   kill $(cat $pidFile)
   sleep 5
   set -e
   rm $pidFile
}

if [[ -f $pidFile ]]; then
   stop_server

   echo 'Destroying old server directory.'
   rm -rf $serverDir
fi

echo 'Unpacking new server dist.'
unzip -q $serverZip -d ~/

start_server
```

As you can see, this bash script starts the application in the background and stores the process id to file. It can then stop the application based on the $pidFile.

Note: This is a very basic example implementation. Ideally, you would use a tool such as systemd, to start/stop your application.



Final step: Deploy!

We've made it! Now it should simply be a matter of two commands:

1. `gradle build`
2. `gradle deploy`

If everything worked out the command prompt/terminal should have output like:

```
webServer#7|/home/webadmin
webServer#9|Stopping webserver (PID = 30)
webServer#9|Destroying old server directory.
webServer#9|Unpacking new server dist.
webServer#9|Starting webserver...
webServer#9|PID is 150
```

We should now be able to open a web browser and browse to "http://webapp-server-ip:8888"
If using our Docker container: "http://localhost:8888"

If the response is "hello world!", then everything worked out!

Bonus step: Testing updates

In webapp/src/main/java/server/Main.java change “hello world!” to “hello universe!” Then build, and deploy again. When browsing to “http://webapp-server-ip:8888” you should now receive “hello universe!”

Now when we make changes to our web server, we no longer have to login to the remote server, stop the application, upload the new code, unpackage it, and rerun. Just build/deploy from command line, or any IDE with gradle support.