



How to Remove Dependencies to Speed Up Release Time

Noah Kruswicki
Software Development Engineer in Test (SDET) III
Accusoft





Shifting Sands

One of the challenges facing modern software development is what I have often heard called “shifting sands.” It means the differing pace of revision in the software you rely on.

Sometimes revisions can impact your sprints. For our specific case in SaaS QA, the fast pace of Chrome and Firefox development means putting in a request with IT to update a Teamcity agent, who would then prioritize it for a future sprint.

This could mean waiting weeks for changes, and in one case, we had fallen behind 2 versions of Chrome and our browser was no longer compatible with the Chrome driver.

Now expand this problem to all the other dependencies we had such as NodeJS, Java, OS version, etc. and you quickly have a maintenance nightmare.

To solve this issue, I started exploring the use of running our test suite in a Docker container so that we only needed an updated Teamcity agent when Docker updated.

Docker Container Testing

After implementing the solution, I had a dockerfile that looked something like this:

```
# Set the base image
```

```
FROM ubuntu
```

```
# wget
```

```
RUN apt-get update
```

```
RUN apt-get install wget -y
```

```
# Chrome
```

```
RUN apt-get install gnupg -y
```

```
RUN wget -q -O - https://dl-ssl.google.com/linux/linux_signing_key.pub | apt-key add -
```

```
RUN sh -c 'echo "deb [arch=amd64] http://dl.google.com/linux/chrome/deb/ stable main" >> /etc/apt/sources.list.d/google-chrome.list'
```

```
RUN apt-get update
```

```
RUN apt-get install google-chrome-stable -y
```

```
# Add non-root user
```

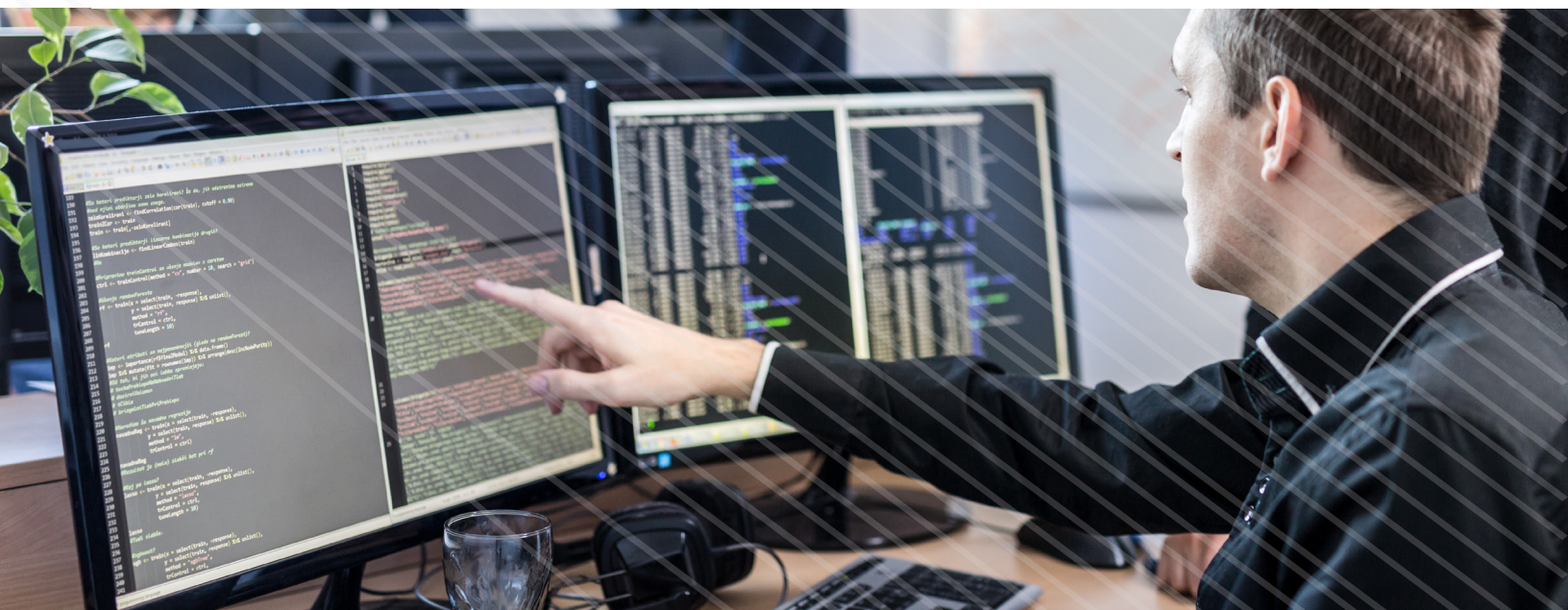
```
RUN useradd --create-home -s /bin/bash tester
```

```
WORKDIR /home/tester
```

```
USER tester
```

```
# NVM
```

```
RUN wget -qO- https://raw.githubusercontent.com/creationix/nvm/v0.33.11/install.sh | bash
```




To run my tests on Docker in Teamcity, I used the Docker plugin/runner. The first step was to build the container from my dockerfile:

Build Step (1 of 2): Build Docker Image | ▾

+ Add

Runner type:

Docker Build  ▾
Runner for Docker files

Step name:

Build Docker Image
Optional, specify to distinguish this build step from other steps.

Execute step: ⓘ


If all previous steps finished successfully ▾
Specify the step execution policy.

Docker Build Parameters



Dockerfile source: ⓘ

File ▾

Path to file: *

Dockerfile  
The specified path should be relative to the checkout directory.

Context folder:

 
If blank, the folder containing the Dockerfile will be used.

Name:tag(s)

jolt-test 
Newline-separated list of the image name:tag(s).

Additional arguments for
'build' command:

--pull 
Additional arguments that will be passed to the 'build' command.

 Hide advanced options

Save

Cancel

Next I needed to run my tests. I used a command line step and told Teamcity to run my tests in the container I built in the previous step and run them with the user 'tester.'

Build Step (2 of 2): Gather Dependencies and Run Tests | ▼ + Add build step »

Runner type: ▼
Simple command execution

Step name:
Optional, specify to distinguish this build step from other steps.

Execute step: ▼
Specify the step execution policy.

Working directory:
Optional, set if differs from the checkout directory.

Run: ▼

Custom script: *
Enter build script content:

```
sh ./tc-run.sh
```


A platform-specific script, which will be executed as a .cmd file on Windows or as a shell script in Unix-like environments.

Format stderr output as: ▼
Specify how error output is processed.

Docker Settings

Run step within Docker container: ▼
E.g. ruby:2.4. TeamCity will start a container from the specified image and will try to run this build step within this container.

Pull image explicitly: If enabled, docker pull <imageName> will be run before docker run command.

Additional docker run arguments:
Edit arguments:

```
-u tester
```


Default argument is --rm, you can specify additional ones.

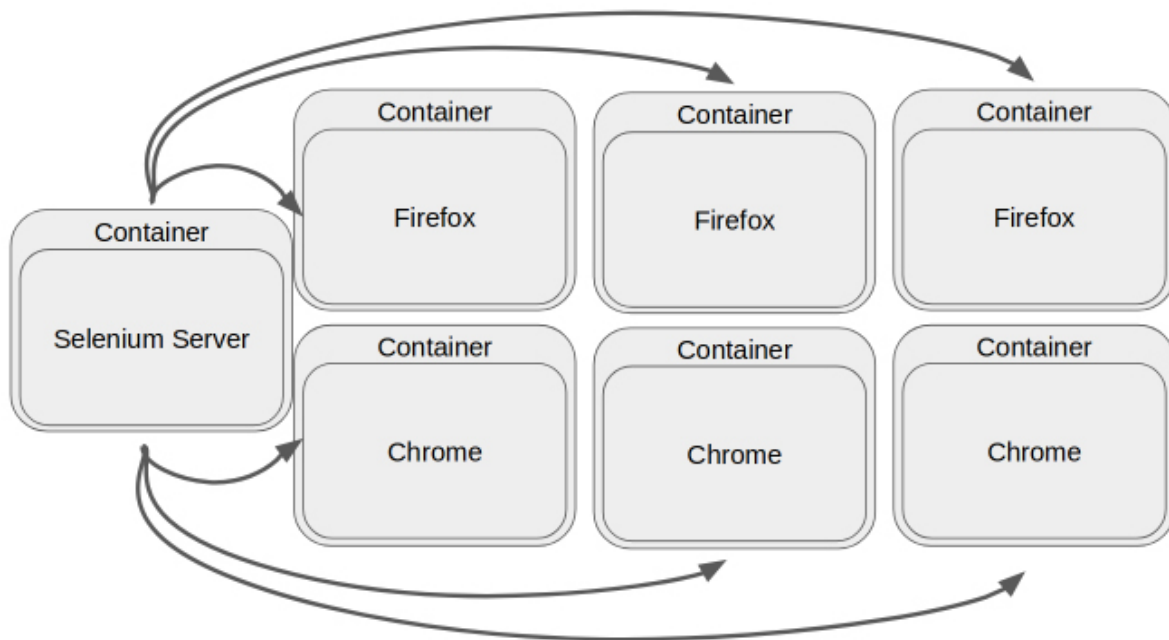
[Hide advanced options](#)

At this point, I was feeling pretty good. Problem solved. Job well done. Go home, everybody. Unfortunately, when presenting this to my boss, he reminded me that one of the specifications for the UI regression tests were that they must run in parallel. Back to the drawing board.

Parallelizing Tests with Docker Compose and Selenium Grid

I knew parallelism could be achieved from Selenium Grid, but I had little experience working with it. After doing some research, I found that it works in a server/client architecture. There was a primary Selenium Grid server that exposed port 4444 for user communication and port 5555 for node communication. After nodes had been configured, they called out to the server to make it aware of the new node. During test runs, the tests come in to the server, which then delegates them out to available nodes, queuing tests when all nodes are in use.

However, as already explained, using Selenium Server in its classic bare metal or even virtualized implementations would only exacerbate the IT bottleneck. Once again, Docker to the rescue. I began writing several dockerfiles and having Docker Compose build them to implement the structure listed below.



After several failed attempts, I found that the Selenium project had already published blessed containers on Docker Hub. From here, it was just a matter of creating my docker-compose.yml:

```
version: '2'
```

```
services:
```

```
hub:
```

```
image: selenium/hub
```

```
environment:
```

- GRID_MAX_SESSION=10
- GRID_TIMEOUT=180000
- GRID_BROWSER_TIMEOUT=180000
- GRID_HUB_PORT=4444

```
ports:
```

- "4444:4444"

```
container_name: hub
```

```
chrome:
```

```
image: selenium/node-chrome
```

```
depends_on:
```

- hub

```
environment:
```

- HUB_HOST=hub
- HUB_PORT=4444
- DBUS_SESSION_BUS_ADDRESS=/dev/null

```
shm_size: 1024MB
```

```
volumes:
```

- /dev/shm:/dev/shm

```
firefox:
```

```
image: selenium/node-firefox
```

```
depends_on:
```

- hub

```
environment:
```

- HUB_HOST=hub
- HUB_PORT=4444
- DBUS_SESSION_BUS_ADDRESS=/dev/null

```
shm_size: 1024MB
```

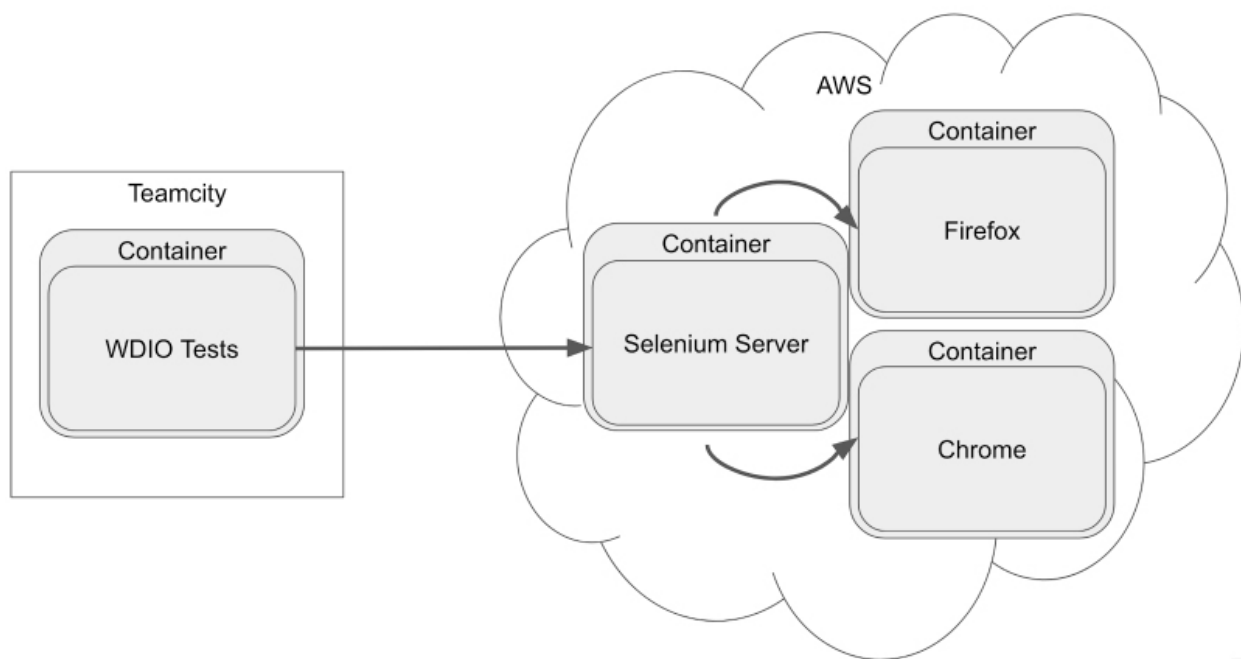
```
volumes:
```

- /dev/shm:/dev/shm

Once I was able to get this working, I tried to migrate it to Teamcity. Unfortunately, due to what I believe to be a bug in Teamcity's Docker implementation, I was not able to get this to work as a self contained unit. I knew this was the right way to go, so I just had to find a way to make it work. I thought that I would spin up a VM in an attempt to host the Selenium Grid containers there. Then, I would point my tests to port 4444 on my VM and attempt to run them from inside a container on Teamcity. SUCCESS!!

Integrating AWS into the Build Pipeline

Now that the dockerized Selenium Grid concept was working, my next step was to find a server somewhere to have the Grid containers running when my tests were ready. I knew from experience running this configuration locally, I would need at least 2 CPUs and at least 8 GBs of RAM. When gathering data, it became clear that this would be a costly server to have sitting in a rack somewhere collecting dust most of the day. So, I thought spinning up an AWS instance right before run time and then spinning it back down after the test runs seemed like the best option.



In order to spin up the AWS instances in an automated way, I needed to use the awscli tool that Amazon publishes. I used this in combination with a shell script. At Accusoft, we use EC2 tags to determine billing for products groups, so including tags was very important. While often times we had the same tags on the same images, on occasion we needed to change the tags. This meant my script had to be easy and fast enough to hit the typical use case, but also needed the control to change the tags on the fly if needed. One additional challenge I faced was the need to pass an instance ID received back from the spin up script to the spin down script.

For this, I created a temporary text file to store the number:

```
#!/bin/bash
PROGNAME=$0
manpage=false
team=""
key=""
secret=""
# Instance Info
instanceType="t3.large"
amiId="ami-xxxxxxxxxx"
keyName="Saas-QA"
securityGroupsIds="sg-xxxxxxxxxx sg-xxxxxxxxxx sg-xxxxxxxxxx"
subnetId="subnet-xxxxxxxxxx"
# Tags
name=""
productGroup=""
rules="none"
environment=""
owner=""
region=""
purpose=""

help() {
  $manpage=true
  cat << EOF >&2
```

Usage: \$PROGNAME [options]

- g <product group> Must be an entry from our approved list of groups. Used for billing.
- r <rules> JSON object with a list of restrictions on the object.
- e <environment> Must be one of the following: prod,dev,qa,staging,support.
- O <owner> Engineering team or specific user who started the instance.
- l <region> AWS region. Most common use is us-east-1.
- p <purpose> String value outlining the purpose of this AWS instance.
- t <team> Team is a predefined set of tags based on known usages.
- k <key> Amazon AWS access key.
- s <secret key> Amazon AWS secret access key.

```

EOF
exit 1
}

setTeam() {
  case $team in
    team1)
      name="team1-automation"
      productGroup="pg1"
      rules="disabled"
      environment="qa"
      owner="owner1"
      region="us-east-1"
      purpose="purpose1"
      ;;
    team2)
      name="team2-automation"
      productGroup="pg2"
      rules="disabled"
      environment="dev"
      owner="owner2"
      region="us-east-1"
      purpose="purpose2"
      ;;
  esac
}

# Override for rule breakers
if [ -z $owner ]
then
  owner=$USER
fi

if [ -z $productGroup ]
then
  productGroup=$TEAMCITY_BUILDCONF_NAME
fi

while getopts g:r:e:O:l:p:t:k:s:o do
  case $o in
    (g) group=$OPTARG;;
    (r) rules=$OPTARG;;
    (e) environment=$OPTARG;;
    (O) owner=$OPTARG;;
    (l) region=$OPTARG;;
    (p) purpose=$OPTARG;;
    (t) team=$OPTARG; setTeam;;
    (k) key=$OPTARG;;
    (s) secret=$OPTARG;;
    (*) help; exit 2;;
  esac
done
shift "$((OPTIND - 1))"

```

```
rm instanceId.txt
```

```
export AWS_ACCESS_KEY_ID=$key  
export AWS_SECRET_ACCESS_KEY=$secret
```

```
aws ec2 run-instances \  
--image-id $amiId \  
--count 1 \  
--instance-type $instanceType \  
--key-name $keyName \  
--security-group-ids $securityGroupsIds \  
--subnet-id $subnetId \  
--region $region \  
--private-ip-address xxxxxxxxxx \  
--tag-specifications \  
"ResourceType=instance,Tags=[{Key=Name,Value=${name}}, \  
{Key=Product Group,Value=${productGroup}}, \  
{Key=Rules,Value=${rules}}, \  
{Key=Environment,Value=${environment}}, \  
{Key=Owner,Value=${owner}}, \  
{Key=Region,Value=${region}}, \  
{Key=Purpose,Value=${purpose}}]" \  
| grep InstanceId | awk '{print $2}' | tr -d , | tr -d \"\`\" >> instanceId.txt
```



Then to spin down the AWS instances I used a script similar in structure:

```
#!/bin/bash
PROGNAME=$0
manpage=false
key=""
secret=""

help() {
  $manpage=true
  cat << EOF >&2

Usage: $PROGNAME [options]

-k <key>           Amazon AWS access key.

-s <secret key>   Amazon AWS secret access key.

EOF
  exit 1
}

while getopts gr:eO:l:p:t:k:s: o; do
  case $o in
    (k) key=$OPTARG;;
    (s) secret=$OPTARG;;
    (*) help; exit 2;;
  esac
done
shift "$((OPTIND - 1))"

export AWS_ACCESS_KEY_ID=$key
export AWS_SECRET_ACCESS_KEY=$secret
export AWS_DEFAULT_REGION=us-east-1
id=$(<instanceId.txt)
aws ec2 terminate-instances --instance-ids $id
```

Starting Selenium Grid Containers

Now that spinning up and down the instances was complete, I needed to find a way to start the Selenium Grid containers when the instances started up. For that, I use a systemd service to kick off the Docker Compose script. Systemd is fairly new on the Linux scene, but it makes it really easy to start services and to tie them to others. So in this case, I didn't want the Selenium services to start before the network or docker services. One quick line, and this is no longer a concern:

```
[Unit]
```

```
Description=Docker compose script for selenium grid
```

```
After=network.target docker.service
```

```
[Service]
```

```
Type=simple
```

```
ExecStart=/usr/local/bin/docker-compose -f /home/weyenk/docker-compose.yml up --scale chrome=10 --scale firefox=10
```

```
ExecStop=/usr/local/bin/docker-compose -f /home/weyenk/docker-compose.yml down
```

```
[Install]
```

```
WantedBy=multi-user.target
```





By adding Docker and Selenium Grid we are now able to overcome impediments that used to delay our releases, sometimes by weeks. As a side benefit, I was able to add visual regression testing to our test arsenal. Additionally, I could now expand our UI based tests to cover additional browsers and increase the parallelism of our tests all while maintaining the confidence our IT staff has in our systems.

In conclusion, the addition of spin up and spin down scripts for AWS makes these tests much cheaper to run in a fiscal sense, which makes any executive happy. Most importantly the success of these infrastructure changes should be even better Accusoft products going forward.