Building a Banner Application

Implementing a Banner Application with Docker

Marianne Gillfillan, Senior DBA & Cloud Architect

October 2022

Strata Information Group
3935 Harney St., Suite 203
San Diego, CA 92110

Contents

1. Building a Banner Application

In this exercise we will build a Docker container to run a Banner application. One of the nice things about containers is that they are very portable and are easily repeatable. Once we get one good working container for one Banner application, we'll be able to copy it for just about all of the other Banner applications we know about with little modifications.

We're going to use applicationNavigator as our example since it is pretty low weight.

Let's start by preparing the work space by creating a directory for all of our files and then change to that directory.

```
mkdir applicatonNavigator
cd applicationNavigator
```

1.1. What do we know about Banner apps?

Just about all Banner applications need certain things in their Tomcat environments:

- jarfiles
- Database connections - although not all applications need the same datasources
- Timezone settings
- Logging settings
- SSL configuration, if desired
- Setting up to run as service

Having each application in a self-contained environment isolates it from anything else in the network. If it gets compromised, just destroy it and recreate it. The container is not meant to last, so don't get attached to it. It is meant to be disposable.

We're going to need a Dockerfile for our Banner application and we'll use our custom tomcat image that we just created:

```
nano Dockerfile
```

Dockerfile

```
FROM mytomcat:8.5.82-2
```

1.2. Building the Dockerfile

What should our Dockerfile start to look like? What are we going to need?

- tomcat image
- war files
- jar files

Dockerfile

```
FROM mytomcat:8.5.82-2

#COPY webapps/*.war /usr/local/tomcat/webapps

# The ojdbc_ver will depend on the version of the database. This can be
 ↪  updated as the database
# is updated to maintain compatibility.
ENV ojdbc_ver=19.10.0.0 \
    ojdbc_url=https://repo1.maven.org/maven2/com/oracle/database \
    tomcat_lib=/usr/local/tomcat/lib

ADD --chown=tomcat:tomcat
 ↪  ${ojdbc_url}/jdbc/ojdbc8/${ojdbc_ver}/ojdbc8-${ojdbc_ver}.jar
 ↪  ${tomcat_lib}/ojdbc8.jar
ADD --chown=tomcat:tomcat
 ↪  ${ojdbc_url}/xml/xdb/${ojdbc_ver}/xdb-${ojdbc_ver}.jar
 ↪  ${tomcat_lib}/xdb.jar
ADD --chown=tomcat:tomcat
 ↪  ${ojdbc_url}/jdbc/ucp/${ojdbc_ver}/ucp-${ojdbc_ver}.jar
 ↪  ${tomcat_lib}/ucp.jar
```

What do we notice about the copying of the war files? We're going to need a webapps directory in working space. Let's create that.

```
mkdir webapps
```

When ready, the appliationNavigator.war can be copied into this webapps directory so when the container is built, the war file will exist inside the container (provided the comment has been removed in the Dockerfile - until then, we're going to leave that commented).

What else can we add to the tomcat environment that might be application specific?

- environment variables
- datasources
- JAVA_OPTS

- perhaps something to process the environment variables

This is where things get a little interesting and totally customizable. The easy way to handle what we're about to do would be to have a local copy of the server.xml and context.xml in the working directory, then use docker COPY commands to add them to the container. While that would be easy, it would also be a security risk as the server.xml contains a plain text password. Not a real good plan if you're pushing this to a code repository of some sort. Best practice is to NEVER store secrets in a code repository.

1.3. Building the datasource

So how do we get around copying entire files with sensitive data? We use environment variables wherever possible and script the rest. So what are the scripting options? Frankly, there are two popular options: bash and ansible. Since ansible would be a workshop in its own rite, we're going to concentrate on a bash solution in this workshop. Please note, there are many ways to skin a cat. So there are many ways to approach what we're about to do. This is just one, very feeble approach.

Let's start with some environment variables related to the datasources that we know we need to add to the server.xml and context.xml files.

Dockerfile

```
ENV TCDS_BP_USER="banproxy" \
    TCDS_BP_JDBC_URL="jdbc:oracle:thin:@host:port/dbservice" \
    TCDS_BP_JNDI_NAME="jdbc/bannerDataSource"

ENV TCDS_SS_USER="ban_ss_user" \
    TCDS_SS_JDBC_URL="jdbc:oracle:thin:@host:port/dbservice" \
    TCDS_SS_JNDI_NAME="jdbc/bannerSsbDataSource"
```

The environment variables can be anything. Note the continuation character ****. Each command in the Dockerfile will create a new layer in the image. The more layers, the bigger the image. To create smaller images, create fewer layers, combine like commands whenever possible. Environment variables are a great place to reduce the number of layers in an image.

Once the environment variable exists, they can be used anywhere in the container, most notably in scripting languages, like bash scripts. We know we need to add datasource information to the server.xml. We know most of it is static information. Let's start with a template.

templates/resource.xml

```
<Resource name="jdbcDatasource"
auth="Container"
type="javax.sql.DataSource"
driverClassName="oracle.jdbc.OracleDriver"
url="jdbcUrl"
username="datasourceUser"
password="datasourcePswd"
initialSize="25"
maxIdle="10"
maxTotal="100"
maxWaitMillis="30000"
minIdle="10"
accessToUnderlyingConnectionAllowed="true"
removeAbandonedOnBorrow="true"
testOnBorrow="true"
testWhileIdle="true"
timeBetweenEvictionRunsMillis="1800000"
validationQuery="select * from dual"
validationQueryTimeout="300"/>
```

We also know the datasource will need a corresponding entry in the context.xml. Let's create a template for that as well.

templates/resourceLink.xml

```
<ResourceLink global="jdbcDatasource" name="jdbcDatasource"
 ↪  type="javax.sql.DataSource"/>
```

Now we should be able to create any datasource we need for any Banner application at any time by simply setting the right environment variables and parsing the template correctly. This is where it gets creative as there are many ways to parse text files.

Let's start building our bash script to parse the templates by creating a ***run.sh*** script.

First we need a way out in case of failure, and of course backup the originals.

run.sh

```
#!/bin/bash

die () {
        echo "ERROR: $*"
        exit 1
}
#
# Backup files
#
for file in server.xml context.xml
do
  cp /usr/local/tomcat/conf/$file /usr/local/tomcat/conf/${file}_$(date
   ↪  '+%Y-%m-%d') || die "Error backing up $file"
  cp /usr/local/tomcat/conf/$file /tmp/$file || die "Error prepping file
   ↪  $file for update"
done
```

Now let's gather the datasource environment variables that were set by the Dockerfile commands:

run.sh

```
#
# Gather Tomcat Datasource variable information
#
read -d "\n" -a ds_array <<< `env |grep "JNDI_NAME"|awk '{print $1}'`
for var in "${ds_array[@]}"
do
  varname=`echo "$var"|awk -F= '{print $1}'`
  varvalue=`echo "$var"|awk -F= '{print $2}'`
  varprefix=`echo "$var"|sed 's/TCDS_//'|sed 's/_.*//'`

  uservar="TCDS_${varprefix}_USER"
  jndivar="TCDS_${varprefix}_JNDI_NAME"
  urlvar="TCDS_${varprefix}_JDBC_URL"
  pswdvar="TCDS_${varprefix}_PASSWORD"
```

Now that we have our working variables all set, we can parse the templates for the variables that exist. Notice, we started a "for-do" loop but we have not ended it yet. All of this is happening for each existence of a variable that that has JNDI_NAME in its name.

Let's update the server.xml with some sed magic

run.sh

```
#
# Update server.xml
#
  if [[ -f "/tmp/resource_${varprefix}.sed" ]]; then
     rm /tmp/resource_${varprefix}.sed
  fi
  cat <<EOF > /tmp/resource_${varprefix}.sed
s|jdbcDatasource|${!jndivar}|
s|jdbcUrl|${!urlvar}|
s|datasourceUser|${!uservar}|
s|datasourcePswd|${!pswdvar}|
EOF

  sed -f /tmp/resource_${varprefix}.sed /tmp/resource.xml
   ↪  >/tmp/resource_${varprefix}.xml
  sed -i '/<\/GlobalNamingResources>/e cat
   ↪  /tmp/resource_'"${varprefix}"'.xml' /tmp/server.xml || die "Error
   ↪  updating server.xml"
```

Let's do likewise for the context.xml and end the "for-do" loop

run.sh

```
#
# Update context.xml
#
  if [[ -f "/tmp/resourceLink_${varprefix}.sed" ]]; then
     rm /tmp/resourceLink_${varprefix}.sed
  fi
  cat <<EOF >> /tmp/resourceLink_${varprefix}.sed
s|jdbcDatasource|${!jndivar}|g
EOF

  sed -f /tmp/resourceLink_${varprefix}.sed /tmp/resourceLink.xml
   ↪  >/tmp/resourceLink_${varprefix}.xml
  sed -i '/<\/Context>/e cat /tmp/resourceLink_'"${varprefix}"'.xml'
   ↪  /tmp/context.xml || die "Error updating context.xml"

done
```

Now we can copy the updated files back into the tomcat directory.

run.sh

```
#
# Copy updated files back to tomcat directory
#
cp /tmp/server.xml /usr/local/tomcat/conf/server.xml || die "Error copying
 ↪  server.xml to tomcat directory"
cp /tmp/context.xml /usr/local/tomcat/conf/context.xml || die "Error copying
 ↪  context.xml to tomcat directory"
```

There's one more thing we'll add to our run script, so we'll set that aside for now.

1.4. Setting up JAVA_OPTS

We also know the setenv.sh needs to be updated with the appropriate JAVA_OPTS. We can use environment variables to add that information as well.

Dockerfile

```
ENV TOMCAT_JAVA_HOME="/usr/local/openjdk-8" \
    TOMCAT_ROOT="/usr/local/tomcat" \
    TOMCAT_JAVA_OPTS="-Djava.awt.headless=true
↪   -Duser.timezone=America/Chicago" \
    TOMCAT_CATALINA_OPTS="-Xms2G -Xmx4G
↪   -Doracle.jdbc.autoCommitSpecCompliant=false
↪   -Dbanner.logging.dir=/app_logs
↪   -Djava.security.egd=file:/dev/../dev/urandom -server -XX:+UseParallelGC"
```

Let's create the template of the ***setenv.sh***

templates/setenv.sh

```
JAVA_HOME="${TOMCAT_JAVA_HOME}"; export JAVA_HOME
CATALINA_HOME="${TOMCAT_ROOT}"; export CATALINA_HOME
JAVA_OPTS="${TOMCAT_JAVA_OPTS}"; export JAVA_OPTS
CATALINA_OPTS="${TOMCAT_CATALINA_OPTS}"; export CATALINA_OPTS
CATALINA_PID="${CATALINA_HOME}/pid"; export CATALINA_PID
```

Now we can finish our ***run.sh*** script and startup tomcat

run.sh

```
#
# Copy setenv.sh to tomcat directory
#
cp /tmp/setenv.sh /usr/local/tomcat/bin/setenv.sh || die "Error copying
↪  setenv.sh to tomcat directory"

#
# Startup tomcat
#
cd $CATALINA_HOME || die "failed to cd to CATALINA_HOME ($CATALINA_HOME)"

xtail ${APP_LOGS} &

bin/catalina.sh run
```

Let's update the Dockerfile now to make sure our templates get into the container and then finish off the rest of the Dockerfile to expose the tomcat port and run our script.

Dockerfile

```
COPY --chown=tomcat:tomcat templates /tmp/
COPY --chown=tomcat:tomcat run.sh /run.sh

RUN chmod +x /run.sh

SHELL ["/bin/bash", "-c"]

EXPOSE 8080

CMD /run.sh
```

1.5. Ready to build Banner App

We're now ready to test out the build of our first Banner application. Before we start the build, what do we know from the start?

1. There is no war file, so the container should start pretty quickly
2. We should see errors connecting the datasource since we have not specified any real database information

So as long as we're aware of those two things, we should be able to deal with any other errors we come across. Let's get building...

We still have our build script that we created while building our custom tomcat image. Let's use that to build our Banner application.

```
. ../build.sh applicationNavigator appnav 3.7 8081
```

To review the log file while this is building

```
docker logs applicationNavigator -f
```

Let's review a few things to make sure we did things correctly

```
Server version name:   Apache Tomcat/8.5.82
Java Home:             /usr/local/openjdk-8/jre
JVM Version:           1.8.0_342-b07
Command line argument: -Duser.timezone=America/Chicago
Command line argument: -Xms2G
Command line argument: -Xmx4G
Command line argument: -Doracle.jdbc.autoCommitSpecCompliant=false
Command line argument: -Djava.security.egd=file:/dev/../dev/urandom
Command line argument: -XX:+UseParallelGC
Command line argument: -Dbanner.logging.dir=/app_logs
```

It looks like all the setenv.sh settings are working as expected and the version of tomcat is correct. How can we check the server.xml and context.xml? We'll have to get into the container to actually verify those files. Let's jump into the container.

/usr/local/tomcat/conf/context.xml

```
docker exec -it applicationNavigator /bin/bash

tomcat@f12d41ec122f:/usr/local/tomcat$ cat conf/context.xml|grep ResourceLink
<ResourceLink global="jdbc/bannerDataSource" name="jdbc/bannerDataSource"
 ↪  type="javax.sql.DataSource"/>
<ResourceLink global="jdbc/bannerSsbDataSource"
 ↪  name="jdbc/bannerSsbDataSource" type="javax.sql.DataSource"/>
```

Great! The context.xml looks perfect. Let's check out the server.xml

/usr/local/tomcat/conf/context.xml

```
docker exec -it applicationNavigator /bin/bash

tomcat@d38d967921bb:/usr/local/tomcat$ cat conf/server.xml

  url="jdbc:oracle:thin:@host:port/dbservice"
  username="banproxy"
  password=""
```

Hmm. Everything looks great, but we have no password in the file. Did we forget something? Since we do not want to store secrets in anything that may end up in a code repository, we need to get creative in how we pass them in. While they can be set as environment variables, we don't necessarily want to add them to the Dockerfile in plain text. That would be bad. If we were doing this in AWS, we could add the secret to Secrets Manager and reference it that way. OCI via Kubernetes has a built-in as well. Docker does have a secrets manager, but you have to enable Docker swarm in order to utilize it. If you're not ready to orchestrate your containers in that way, you need to find a way to systematically and securely pass secrets into your container. Here's one way to do that.

1.6. Handling Secrets

As we begin to think ahead at all the different types of applications we'll need to start up, not all of them require the same datasources. While many of them do, not all of them do. For now, we may want to have a custom build script for each application. So, let's make a copy of the build.sh script and copy it to our applicationNavigator directory. Make sure you're in the right directory before proceeding.

```
cd applicationNavigator
cp ../build.sh .
```

Let's add a few things to the build script. First, we'll need to prompt for the password information.

build.sh

```
if [ ! -n "$bppw" ]; then read -sp 'Enter banproxy password: ' bppw; echo "";
↪  fi
if [ ! -n "$sspw" ]; then read -sp 'Enter ban_ss_user password: ' sspw; echo
↪  ""; fi
```

Now that we have the values, we'll need to do something with them. We can pass environment variables on the docker command line by modifying the docker run statement. Also, remember to unset the variables so we don't leave passwords lingering around.

build.sh

```
[ -n "$bppw" ] && [ -n "$sspw" ] && docker run --name $CNTRNAME -p
↪  $PORTNUM:8080 -d -e TCDS_BP_PASSWORD=$bppw -e TCDS_SS_PASSWORD=$sspw
↪  $APPIMG:$APPVER

unset bppw
unset sspw
```

Let's try to start up our container.

```
. build.sh applicationNavigator appnav 3.7 8081
Enter banproxy password:
Enter ban_ss_user password:
```

Notice, we are prompted for the passwords, but they are not echoed to the terminal. While this may not be an ideal solution, it will get us moving forward for now.

With the container started up, let's check to make sure the environment variables got created as expected.

```
env|grep PASS
TCDS_BP_PASSWORD=thisisonevalue
TCDS_SS_PASSWORD=thisissomethingelse
```

Also, verify they were added to the server.xml file properly.

conf/server.xml

```
username="banproxy"
password="thisisonevalue"

username="ban_ss_user"
password="thisissomethingelse"
```

However you chose to handle secrets, you now know it is possible to process them through your container. Just be vigilant about how you do it and never store secrets in a code repository.

1.7. Persistent volumes

We mentioned previously that containers are meant to be disposal. As such, when the container is terminated, so are all the files that are stored within it. This means that all the data is lost. Have no fear. If there is ever a need for data to persist past the termination of the container, we can attach a host known device and reference that device inside the container as a volume.

What Banner applications can we think of that would require persistent data?

- Banner Extensibilty - pbroot directory location
- Student Self Service - extensions and photo images
- Faculty Self Service - extensions
- Employee Self Service - photo images
- Banner 8 Self Service - photo images
- Admin Pages - photo images
- Ethos API Management Center - config settings

Some volumes we attach will need to be read only and some will need to have read/write access. In the above list, can you identify which volumes would need which type of access?

Volume parameter can be passed on the docker command line or can be created via the docker volume command. For the purpose of this workshop, we'll be passing it as a parameter on the command line using the something like the following:

```
# Read only parameter
-v /mnt/images:/img/photos:ro

# Read/write volume parameter
-v /home/devops/eamc:/app_config
```

While there is no functional need to have a persistent volume on the applicationNavigator container, we'll use it anyway as our example just to prove that we can do it.

Since we have locked down the container to the run as the tomcat user, we will not be able to create any new directories. We can get around that by using the COPY command and copying a stub directory from the host.

A stub directory is something that is only known to the container upon container creation time and is pulled in from an alternate location. In this case, since the tomcat image is locked down from creating directories under certain locations and we know the the docker COPY command can carry ownership privileges with it, we can initially create the directories we need with the ownership of the tomcat user so the application can act upon the directory space.

Let's create two stub directories, one for testing the read only functionality and one for testing the read/write access. Keep in mind, these are just stub directories and not the real volumes that will

be mounted to the container. We just these directories so we can create the devices inside the container. Just as the directories need to exist before we mount a device on a typical OS system, so too do they need to exist in a container before we attempt to mount them.

```
cd applicationNavigator
mkdir test-ro
mkdir test-rw
```

With our directories created, let's add the COPY statements to our Dockerfile.

Dockerfile

```
COPY --chown=tomcat:tomcat test-ro /test-ro
COPY --chown=tomcat:tomcat test-rw /test-rw
```

We want to make sure the user tomcat owns the directories and that the user can read and write to it, if necessary.

We need to create the source volumes that we plan to mount to the containers and make sure they have some files in them for testing purposes.

```
sudo mkdir /mnt/img/test-ro -p
sudo mkdir /mnt/img/test-rw -p
sudo touch /mnt/img/test-ro/file1.txt
sudo echo "The quick brown fox jumped over the lazy dog."
 ↪  >/mnt/img/test-ro/file2.txt
sudo touch /mnt/img/test-rw/file3.txt
sudo touch /mnt/img/test-rw/file4.txt
```

Now let's add the necessary parameters to our docker build statement so we can access the files within the container.

build.sh

```
[ -n "$bppw" ] && [ -n "$sspw" ] && docker run --name $CNTRNAME -p
 ↪  $PORTNUM:8080 -d -e TCDS_BP_PASSWORD=$bppw -e TCDS_SS_PASSWORD=$sspw -v
 ↪  /mnt/img/test-ro:/test-ro:ro -v /mnt/img/test-rw:/test-rw $APPIMG:$APPVER
```

Let's rebuild the container

```
. build.sh applicationNavigator appnav 3.7 8081
```

We can inspect the structure of the container using the docker inspect command. The structure is displayed in json format.

```
docker inspect applicationNavigator
```

```
"Mounts": [
    {
        "Type": "bind",
        "Source": "/mnt/img/test-ro",
        "Destination": "/test-ro",
        "Mode": "ro",
        "RW": false,
        "Propagation": "rprivate"
    },
    {
        "Type": "bind",
        "Source": "/mnt/img/test-rw",
        "Destination": "/test-rw",
        "Mode": "",
        "RW": true,
        "Propagation": "rprivate"
    }
],
```

We should do some testing to verify the volumes are acting as we expect. Let's jump into the container and check things out.

```
docker exec -it applicationNavigator /bin/bash

ls -ld /test*
drwxr-xr-x 2 root root 4096 Sep 25 14:34 /test-ro
drwxr-xr-x 2 root root 4096 Sep 25 14:34 /test-rw

cd /test-ro
touch newfile.txt
touch: cannot touch 'newfile.txt': Read-only file system

rm file2.txt
rm: remove write-protected regular file 'file2.txt'? y
rm: cannot remove 'file2.txt': Read-only file system

cat file2.txt
The quick brown fox jumped over the lazy dog.

cd /test-rw
touch newfile-rw.txt
touch: cannot touch 'newfile-rw.txt': Permission denied

ls -l
total 0
-rw-r--r-- 1 root root 0 Sep 25 14:34 file3.txt
-rw-r--r-- 1 root root 0 Sep 25 14:34 file4.txt
```

Volumes will inherit the file permissions from the host system. So, if you expect the tomcat user to be able to read and/or write the files inside the container, make sure the tomcat user can do so outside the container as well. This will get tricky when dealing with different UID/GID combinations between systems, so having a few sysadmin skills here may come in handy.

```
#Outside the container
ls /mnt/img -Rl
/mnt/img:
total 8
drwxr-xr-x 2 root root 4096 Sep 25 15:13 test-ro
drwxr-xr-x 2 root root 4096 Sep 25 14:34 test-rw

/mnt/img/test-ro:
total 8
-rw-r--r-- 1 root root 86 Sep 25 15:13 file1.txt
-rw-r--r-- 1 root root 46 Sep 25 15:13 file2.txt

/mnt/img/test-rw:
total 0
-rw-r--r-- 1 root root 0 Sep 25 14:34 file3.txt
-rw-r--r-- 1 root root 0 Sep 25 14:34 file4.txt

sudo chown 10000:10001 -R /mnt/img

docker exec -it applicationNavigator /bin/bash

#Insides the container
ls -lR /test*
/test-ro:
total 8
-rw-r--r-- 1 tomcat tomcat 86 Sep 25 15:13 file1.txt
-rw-r--r-- 1 tomcat tomcat 46 Sep 25 15:13 file2.txt

/test-rw:
total 0
-rw-r--r-- 1 tomcat tomcat 0 Sep 25 14:34 file3.txt
-rw-r--r-- 1 tomcat tomcat 0 Sep 25 14:34 file4.txt
```

What other types of sysadmin commands can be run to affect file permissions inside the container? What about chmod to affect rwx permissions? Permissions like these will need to be set appropriately for the files expected to be modified from within the container. It is important to know what the container will be modifying before the volume is mounted.

1.8. Review

We've seen how we can create a Banner application from a command file and a script or two. We've used templates to systematically modify standard files. We explored how to attach volumes when data persistence is required. This example can now be replicated for any Banner application, modifying for specific environmental differences: memory, war files, volumes, users, etc.

But what happens when upgrades come around? We'll address that in the next lesson on ***Updating Images***.