Heroku Database Deployments

jim stafford

Fall 2022 v2021-08-28: Built: 2022-12-07 06:16 EST

Table of Contents

Chapter 1. Introduction

This lecture contains several "how to" aspects of building and deploying a Docker image to Heroku with Postgres or Mongo database dependencies.

1.1. Goals

You will learn:

- how to build a Docker image as part of the build process
- how to provision Postgres and Mongo internet-based resources for use with Internet deployments
- how to deploy an application to the Internet to use provisioned Internet resources

1.2. Objectives

At the conclusion of this lecture and related exercises, you will be able to:

1. provision a Postgres Internet-accessible database
2. provision a Mongo Internet-accessible database
3. map Heroku environment variables to Spring Boot properties using a shell script
4. build a Docker image as part of the build process

Chapter 2. Production Properties

We will want to use real database instances for remote deployment and we will get to that in a moment. For right now, lets take a look at some of the Spring Boot properties we need defined in order to properly make use of a live database.

2.1. Postgres Production Properties

We will need the following RDBMS properties individually enumerated for Postgres at runtime.

- `spring.data.datasource.url`
- `spring.data.datasource.username`
- `spring.data.datasource.password`

The remaining properties can be pre-set with a properties configuration embedded within the application.

Production Properties

```
##rdbms
#spring.datasource.url=... ①
#spring.datasource.username=...
#spring.datasource.password=...

spring.jpa.show-sql=false
spring.jpa.hibernate.ddl-auto=validate
spring.flyway.enabled=true
```

① datasource properties will be supplied at runtime

2.2. Mongo Production Properties

We will need the Mongo URL and luckily that and the user credentials can be expressed in a single URL construct.

- `spring.data.mongodb.uri`

There are no other mandatory properties to be set beyond the URL.

Production Properties

```
#mongo
#spring.data.mongodb.uri=mongodb://... ①
```

① mongodb.uri — with credentials — will be supplied at runtime

Chapter 3. Parsing Runtime Properties

The Postgres URL will be provided to us by Heroku using the `DATABASE_URL` property as show below. They provide a means to separate the URL into variables, but that feature was not available for Docker deployments at the time I investigated. We can easily to that ourselves.

A logically equivalent Mongo URL will be made available from the Mongo resource provider. Luckily we can pass that single value in as the Mongo URL and be done.

Example Input Environment Variables

```
DATABASE_URL=postgres://postgres:secret@postgres:5432/postgres
MONGODB_URI=mongodb://admin:secret@mongo:27017/votes_db?authSource=admin
```

3.1. Environment Variable Script

Earlier — when PORT was the only thing we had to worry about — I showed a way to do that with the Dockerfile `CMD` option.

Review: Turning PORT Environment Variable into server.port Property

```
ENV PORT=8080
ENTRYPOINT ["java", "org.springframework.boot.loader.JarLauncher"]
CMD ["--server.port=${PORT}"]
```

We could have expanded that same approach if we could get the DATABASE_URL broken down into URL and credentials. With that option not available, we can delegate to a script.

The following snippet shows the skeleton of the `run_env.sh` script we will put in place to address all types of environment variables we will see in our environments. The shell will launch whatever command was passed to it ("$@") and append the `OPTIONS` that it was able to construct from environment variables. We will place this in the `src/docker` directory to be picked up by the Dockerfile.

The resulting script was based upon the much more complicated example.

run_env.sh Environment Variable Script

```
#!/bin/bash

OPTIONS=""

#ref: https://raw.githubusercontent.com/heroku/heroku-buildpack-jvm-
common/main/opt/jdbc.sh
if [[ -n "${DATABASE_URL:-}" ]]; then
  # ...
fi
```

```
if [[ -n "${MONGODB_URI:-}" ]]; then
  # ...
fi

if [[ -n "${PORT:-}" ]]; then
  # ...
fi

exec $@ ${OPTIONS}
```

3.2. Script Output

The following snippet shows an example `args` print of what is passed into the Spring Boot application from the run_env.sh script.

Resulting Command Line

```
args [--spring.datasource.url=jdbc:postgresql://postgres:5432/postgres,
--spring.datasource.username=postgres, --spring.datasource.password=secret,
--spring.data.mongodb.uri=mongodb://admin:secret@mongo:27017/votes_db?authSource=admin
]
```

Review: Remember that our environment will look like the following.

Input Environment Variables

```
DATABASE_URL=postgres://postgres:secret@postgres:5432/postgres
MONGODB_URI=mongodb://admin:secret@mongo:27017/votes_db?authSource=admin
```

Lets break down the details.

3.3. Heroku DataSource Property

The following script will breakout URL, username, and password and turn them into Spring Boot properties on the command line.

DataSource Properties

```
if [[ -n "${DATABASE_URL:-}" ]]; then
  pattern="^postgres://(.+):(.+)@(.+)$"  ①
  if [[ "${DATABASE_URL}" =~ $pattern ]]; then ②
    JDBC_DATABASE_USERNAME="${BASH_REMATCH[1]}"
    JDBC_DATABASE_PASSWORD="${BASH_REMATCH[2]}"
    JDBC_DATABASE_URL="jdbc:postgresql://${BASH_REMATCH[3]}"

    OPTIONS="${OPTIONS} --spring.datasource.url=${JDBC_DATABASE_URL} "
    OPTIONS="${OPTIONS} --spring.datasource.username=${JDBC_DATABASE_USERNAME}"
    OPTIONS="${OPTIONS} --spring.datasource.password=${JDBC_DATABASE_PASSWORD}"
```

```
  else
    OPTIONS="${OPTIONS} --no.match=${DATABASE_URL}" ③
  fi
fi
```

① regular expression defining three (3) extraction variables

② if the regular expression finds a match, we will pull that apart and assemble the properties

③ if no match is found, `--no.match` is populated with the DATABASE_URL to be printed for debug reasons

3.4. Testing DATABASE_URL

You can test the script so far by invoking the with the environment variable set.

Testing Postgres URL Parsing

```
(export DATABASE_URL=postgres://postgres:secret@postgres:5432/postgres && bash
./src/docker/run_env.sh echo)
```

Expected Postgres Output

```
--spring.datasource.url=jdbc:postgresql://postgres:5432/postgres
--spring.datasource.username=postgres --spring.datasource.password=secret
```

Of course, that same test could be done with a Docker image.

Testing Postgres URL Parsing within Docker

```
docker run --rm \
-e DATABASE_URL=postgres://postgres:secret@postgres:5432/postgres \①
-v `pwd`/src/docker/run_env.sh:/tmp/run_env.sh \②
openjdk:17.0.2 \
/tmp/run_env.sh echo ③
```

① setting the environment variable

② mounting the file in the `/tmp` directory

③ running script and passing in `echo` as executable to call

3.5. MongoDB Properties

The Mongo URL we get from Atlas can be passed in as a single property. If Postgres was this straight forward, we could have stuck with the `CMD` option.

MongoDB Property

```
if [[ -n "${MONGODB_URI:-}" ]]; then
  OPTIONS="${OPTIONS} --spring.data.mongodb.uri=${MONGODB_URI}"
```

```
fi
```

Demonstrating Mongo URL Handling

```
(export MONGODB_URI=mongodb://admin:secret@mongo:27017/votes_db?authSource=admin &&
bash ./src/docker/run_env.sh echo)
```

Expected Mongo Output

```
--spring.data.mongodb.uri=mongodb://admin:secret@mongo:27017/votes_db?authSource=admin
```

3.6. PORT Property

We need to continue supporting the `PORT` environment variable and will add a block for that.

Server Port Property

```
if [[ -n "${PORT:-}" ]]; then
  OPTIONS="${OPTIONS} --server.port=${PORT}"
fi
```

Testing All Together

```
(export DATABASE_URL=postgres://postgres:secret@postgres:5432/postgres && export
MONGODB_URI=mongodb://admin:secret@mongo:27017/votes_db?authSource=admin && export
PORT=7777 && bash ./src/docker/run_env.sh echo)
```

Expected Aggregate Output

```
--spring.datasource.url=jdbc:postgresql://postgres:5432/postgres
--spring.datasource.username=postgres --spring.datasource.password=secret
--spring.data.mongodb.uri=mongodb://admin:secret@mongo:27017/votes_db?authSource=admin
--server.port=7777
```

Chapter 4. Docker Image

With the embedded properties set, we are now ready to build a Docker image. We will use a Maven plugin to build the image using Docker since the memory requirement for the default Spring Boot Docker image exceeds the Heroku Memory limit for free deployments.

4.1. Dockerfile

The following shows the Dockerfile being used. It is 99% of what can be found in the Spring Boot Maven Plugin Documentation except for:

- a tweak on the `ARG JAR_FILE` command to add our `bootexec` classifier. Note that our local Maven pom.xml `JAR_FILE` declaration will take care of this as well.
- `src/docker/run_env.sh` script added to search for environment variables and break them down into Spring Boot properties

Example Dockerfile

```
FROM openjdk:17.0.2 as builder
WORKDIR application
ARG JAR_FILE=target/*-bootexec.jar ①
COPY ${JAR_FILE} application.jar
RUN java -Djarmode=layertools -jar application.jar extract

FROM openjdk:17.0.2
WORKDIR application
COPY --from=builder application/dependencies/ ./
COPY --from=builder application/spring-boot-loader/ ./
COPY --from=builder application/snapshot-dependencies/ ./
COPY --from=builder application/application/ ./
COPY src/docker/run_env.sh ./ ②
RUN chmod +x ./run_env.sh
ENTRYPOINT ["./run_env.sh", "java","org.springframework.boot.loader.JarLauncher"]
```

① Spring Boot executable JAR has `bootexec` Maven `classifier` suffix added

② added a filter script to break certain environment variables into separate properties

4.2. Spotify Docker Build Maven Plugin

At this point with a Dockerfile in hand, we have the option of building the image with straight `docker build` or `docker-compose build`. We can also use the Spotify Docker Maven Plugin to automate the build of the Docker image as part of the module build. The plugin is forming an explicit path to the JAR file and using the `JAR_FILE` variable to pass that into the `Dockerfile`. Note that by supplying the `JAR_FILE` reference here, we can build images without worrying about the wildcard glob in the Dockerfile locating too many matches.

Spotify Docker Build Maven Plugin

```
<plugin>
    <groupId>com.spotify</groupId>
    <artifactId>dockerfile-maven-plugin</artifactId>
    <configuration>
        <repository>${project.artifactId}</repository>
        <tag>${project.version}</tag>
        <buildArgs>
<JAR_FILE>target/${project.build.finalName}-${spring-boot.classifier}.jar</JAR_FILE>
①
        </buildArgs>
    </configuration>
    <executions>
        <execution>
            <goals>
                <goal>build</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

① JAR_FILE is passed in as a build argument to Docker

Spotify Docker Build Maven Plugin Completing Build

```
[INFO] Successfully built dfe2383f7f68
[INFO] Successfully tagged xxx:6.0.1-SNAPSHOT
[INFO]
[INFO] Detected build of image with id dfe2383f7f68
...
[INFO] Successfully built dockercompose-votes-svc:6.0.1-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
```

Chapter 5. Heroku Deployment

The following are the basic steps taken to deploy the Docker image to Heroku.

5.1. Provision MongoDB

MongoDB offers a Mongo database service on the Internet called Atlas. They offer free accounts and the ability to setup and operate database instances at no cost.

- Create account using email address
- Create a new project
- Create a new (free) cluster within that project
- Create username/password for DB access
- Setup Internet IP whitelist (can be wildcard/all) of where to accept connects from. I normally set that to everywhere — at least until I locate the Heroku IP address.
- Obtain a URL to connect to. It will look something like the following:

```
mongodb+srv://(username):(password)@(host)/(dbname)?retryWrites=true&w=majority
```

5.2. Provision Application

Refer back to the Heroku lecture for details, but essentially

- create a new application
- set the MONGODB_URI environment variable for that application
- set the `SPRING_PROFILES_ACTIVE` environment variable to `production`

```
$ heroku create [app-name]
$ heroku config:set MONGODB_URI=mongodb+srv://(username):(password)@(host)/votes_db...
--app (app-name)
$ heroku config:set SPRING_PROFILES_ACTIVE=production
```

5.3. Provision Postgres

We can provision Postgres directly on Heroku itself.

Example Postgres Provision

```
$ heroku addons:create heroku-postgresql:hobby-dev
Creating heroku-postgresql:hobby-dev on □ xxx... free
Database has been created and is available
 ! This database is empty. If upgrading, you can transfer
 ! data from another database with pg:copy
```

```
Created postgresql-shallow-xxxxx as DATABASE_URL
Use heroku addons:docs heroku-postgresql to view documentation
```

After the provision, we can see that a compound DATABASE_URI was provided

```
$ heroku config --app app-name
=== app-name Config Vars
DATABASE_URL: postgres://(username):(password)@(host):(port)/(database)
MONGODB_URI:  mongodb+srv://(username):(password)@(host)/votes_db?...
SPRING_PROFILES_ACTIVE: production
```

5.4. Deploy Application

Tag Docker Image

```
$ docker tag (artifactId):(tag) registry.heroku.com/(app-name)/web
```

Push Docker Image Using Tag

```
$ heroku container:login
Login Succeeded
$ docker push registry.heroku.com/(app-name)/web
The push refers to repository [registry.heroku.com/(app-name)/web]
6f38c0466979: Pushed
69a39355b3ac: Pushed
ea12a8cf9f94: Pushed
d2451ff7adf4: Layer already exists
...
7ef368776582: Layer already exists
latest: digest:
sha256:21197b193a6657dd5e6f10d6751f08faa416a292a17693ac776b211520d84d19 size: 3035
```

5.5. Release the Application

Invoke the Heroku release command to make the changes visible to the Internet.

Make Application Available

```
$ heroku container:release web --app (app-name)
Releasing images web to (app-name)... done
```

Tail the Heroku log to verify the application starts and the production profile is active.

Tail Heroku Log

```
$ heroku logs --app (app-name) --tail
```

```
/\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
'  |____| .__|_| |_|_| |_\__, | / / / /
=========|_|==============|___/=/_/_/_/
:: Spring Boot ::        (2.7.0)
The following profiles are active: production ①
```

① make sure the application is running the correct profile

Chapter 6. Summary

In this module we learned:

- how to provision internet-based MongoDB and Postgres resources
- how to deploy an application to the Internet to use provisioned Postgres and Mongo database resources
- how to build a Docker image as part of the build process