Heroku Docker Deployments

jim stafford

Fall 2022 v2021-08-26: Built: 2022-12-07 06:18 EST

Table of Contents

Chapter 1. Introduction

With a basic introduction to Docker under our belt, I would like to return to the Heroku deployment topic to identify the downside of deploying full applications — whether they be

- naked Spring Boot executable JAR
- Spring Boot executable JAR wrapped in a Docker image

— and show the benefit of using a layered application.

This is a follow-on lecture

It is assumed that you have already covered the Heroku deployment and Docker lectures, have a Heroku account, already deployed a Spring Boot application, and interacted with that application via the Internet. If not, you will need to go back to that lecture and review the basics of getting started with the Heroku account.

If you do not have Docker — the product — installed, you should still be able to follow along to pick up the concepts.

1.1. Goals

You will learn:

- to deploy a Docker-based image to an cloud provider to make it accessible to Internet users

1.2. Objectives

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

1. make a Heroku-deployable Docker image that accepts environment variable(s)
2. deploy a Docker image to Heroku using docker repository commands
3. deploy a Docker image to Heroku using CLI commands

Chapter 2. Heroku Docker Notes

The following are Heroku references for Spring Boot and Docker deployments

- Developing with Docker
- Working with Spring Boot

Of important note — the Maven Spring Boot Plugin built Docker image (using buildpack) — uses an internal memory calculator that initially mandates 1GB of memory. This exceeds the free 512MB Heroku limit. Deploying this version of the application will immediately fail until we locate a way to change that value. However, we can successfully deploy the standard Dockerfile version — which lacks an explicit, up-front memory requirement.

We will also need to do some property expression gymnastics that will be straight forward to implement using the standard Dockerfile approach.

Chapter 3. Heroku Login

With the Heroku CLI installed — we need to login. This will redirect us to the browser where we can complete the login.

Heroku Command Line Login

```
$ heroku login
heroku: Press any key to open up the browser to login or q to exit:
Opening browser to https://cli-auth.heroku.com/auth/cli/browser/f944d777-93c7-40af-
b772-0a1c5629c609
Logging in... done
Logged in as ...
```

3.1. Heroku Container Login

Heroku requires an additional login step to work with containers. With the initial login complete — no additional credentials will be asked for but this step seems required.

Additional Heroku Container Login

```
$ heroku container
7.60.0
$ heroku container:login
Login Succeeded
```

3.2. Create Heroku App

At this point you are ready to again perform a one-time (per deployment app) process that will reserve an app-name for you on herokuapp.com. We know that this name is used to reference our application and form a URL to access it on the Internet.

Example Create Heroku App

```
$ heroku create [app-name] ①

Creating ⬢ [app-name]... done
https://app-name.herokuapp.com/ | https://git.heroku.com/app-name.git
```

① if app-name not supplied, a random app-name will be generated

Heroku also uses Git repositories for deployment

Heroku creates a Git repository for the app-name that can also be leveraged as a deployment interface. I will not be covering that option.

You can create more than one heroku app and the app can be renamed with the following `apps:rename` command.

Example Rename Heroku App

```
$ heroku apps:rename --app oldname newname
```

Visit the Heroku apps page to locate technical details related to your apps.

Chapter 4. Adjust Dockerfile

Heroku requires the application accept a `$PORT` environment variable to identify the listen port at startup. We know from our lessons in configuration, we can accomplish that by supplying a Spring Boot property on the command line.

Example Startup Specification of Listen Port

```
java -jar (app).jar --server.port=$PORT
```

Since we are launched using a Dockerfile and the parameter will require a shell evaluation, we can accomplish this by using the Dockerfile `CMD` command below — which will feed the `ENTRYPOINT` command its resulting values when expressed this way. [1] I have also added a default value of 8080 when the `$PORT` variable has not been supplied (i.e., in local environment).

Example Spring Boot Dockerfile ENTRYPOINT

```
ENV PORT=8080 ①
ENTRYPOINT ["java", "org.springframework.boot.loader.JarLauncher"] ②
CMD ["--server.port=${PORT}"] ③
```

① default value used if `PORT` is not supplied

② ENTRYPOINT always executes no matter if a parameter is supplied

③ CMD expresses a default when no parameter(s) are supplied

4.1. Test Dockerfile server.port

We can test the configuration locally using the following commands.

4.1.1. Testing $PORT CMD With Environment Variable

In this iteration, we are simulating the Heroku container supplying a `PORT` environment variable with a value. This value will be used by the Spring Boot application running within the Docker image. The `PORT` value is also mapped to an external `9090` value so we can call the server from the outside.

Testing With Environment Variable

```
$ mvn package spring-boot:repackage -Dlayered=true ①

$ docker build . -f Dockerfile.layered -t docker-hello-example:layered
$ docker run --rm -p 9090:5000 -e PORT=5000 docker-hello-example:layered ②
...
Tomcat started on port(s): 5000 (http) with context path '' ③
Started DockerHelloExampleApp in 3.623 seconds (JVM running for 4.392)
```

① example sets layered false by default and toggles with `layered` property

② `-e` option defines `PORT` environment variable to value `5000`

③ `--server.port=${PORT}` sees that value and server listens on port `5000`

We can use the following to test.

Testing $PORT CMDs Mapped to 9090

```
$ curl http://localhost:9090/api/hello?name=jim
hello, jim
```

4.1.2. Testing without Environment Variable

In this iteration, we are simulating local development independent of the Heroku container by not supplying a `PORT` environment variable and using the default from the Docker `CMD` setting. Like before, this value will be used by the Spring Boot application running within the Docker image and that value will again be mapped to external port `9090` value so we can call the server from the outside.

Testing $PORT CMD Without Environment Variable

```
$ docker run --rm -p 9090:8080 docker-hello-example:layered ①
Tomcat started on port(s): 8080 (http) with context path '' ②
Started DockerHelloExampleApp in 4.414 seconds (JVM running for 5.177)
```

① no `PORT` environment variable is expressed

② server uses assigned ENV default of 8080

We can again use the following to test.

Testing $PORT CMDs Mapped to 9090

```
$ curl http://localhost:9090/api/hello?name=jim
hello, jim
```

[1] *"Docker RUN vs CMD vs ENTRYPOINT"*,Yuri Pitsishin, April 2016

Chapter 5. Deploy Docker Image

I will demonstrate two primary ways deploy a Docker image to Heroku:

1. using `docker push` command to deploy a tagged image to the Heroku Docker repository
2. using the `heroku container:push` command to build and upload an image

Both require a follow-on `heroku container:release` command to complete the deployment.

5.1. Deploying Tagged Image

One way to deploy a Docker image to Heroku is to create a Docker tag associated with the target Heroku repository and then push that image to the Docker repository. The tag has the following format

```
registry.heroku.com/[app-name]/web ①
```

① `registry.heroku.com` is the actual address of the Heroku Docker repository

My examples will use the app-name `ejava-docker`.

5.1.1. Tagging the Image

There are at least two ways to tag the image:

- WAY 1: tag the Docker image during the build

  Tag Docker Image During Build

  ```
  docker build . -f Dockerfile.layered -t registry.heroku.com/ejava-docker/web
  ...
  Successfully tagged registry.heroku.com/ejava-docker/web:latest
  ```

- WAY 2: tag an existing Docker image

  Tag Existing Docker Image

  ```
  $ docker build . -f Dockerfile.layered -t docker-hello-example:layered
  $ docker tag docker-hello-example:layered registry.heroku.com/ejava-docker/web
  ```

In either case, we will end up with a tag in the repository that will look like the following.

Example Docker Image Repository with Tagged Image

```
$ docker images | grep heroku
REPOSITORY                            TAG     IMAGE ID      CREATED        SIZE
registry.heroku.com/ejava-docker/web  latest  72fe4327f05f  15 minutes ago 293MB
```

5.1.2. Deploying the Image

The last step in deploying the tagged image is to invoke `docker push` using the full name of the tag.

Push Tagged Docker Image

```
$ docker push registry.heroku.com/ejava-docker/web
The push refers to repository [registry.heroku.com/ejava-docker/web]
3e974fa6054f: Pushed
...
7ef368776582: Layer already exists
latest: digest:
sha256:37c99a899b26f2cfb192cd42f930120b11bb56408eb3e4590dfe78b957f2acf1 size: 2621
```

5.2. Push using Heroku CLI

The other alternative is to use `heroku container:push` to build and push the Docker image without going through the local repository.

```
$ heroku container:push web --app ejava-docker
```

container:push requires Dockerfile to be named Dockerfile — no file references

The `heroku container:push` command requires the Dockerfile be called `Dockerfile` and in the current directory. The command does not allow us to reference a unique filename (e.g., `Dockerfile.layered`). I used a soft link to get around that (i.e., `ln -s Dockerfile.layered Dockerfile`). The `container:push` documentation does infer that files normally referenced locally by the Dockerfile can be in a referenced location — possibly allowing the Dockerfile to be placed in a unique location versus having a unique name.

Chapter 6. Complete Deployment

A successfully pushed image will not be made immediately available. We must follow through with a `release` command.

6.1. Release Pushed Image to Users

The following command finishes the deployment — making the updated image accessible to users.

Release Pushed Image to Users

```
$ heroku container:release web --app ejava-docker
Releasing images web to ejava-docker... done
```

6.2. Tail Logs

We can gain some insight into the application health by tailing the logs.

```
$ heroku logs --app ejava-docker --tail
Starting process with command `--server.port\=\$\{PORT:-8080\}`
...
Tomcat started on port(s): 54644 (http) with context path ''
Started DockerHelloExampleApp in 9.194 seconds (JVM running for 9.964)
```

6.3. Access Site

We can access the deployed application at this point but will be required to use HTTPS. Notice, however, HTTPS is fully setup with a trusted certificate.

Access Deployed Application on Heroku

```
$ curl -v https://ejava-docker.herokuapp.com/api/hello?name=jim
*   Trying 52.73.83.132...
* TCP_NODELAY set
* Connected to ejava-docker.herokuapp.com (52.73.83.132) port 443 (#0)
* SSL connection using TLSv1.2 / ECDHE-RSA-AES128-GCM-SHA256
* ALPN, server did not agree to a protocol
* Server certificate:
*  subject: C=US; ST=California; L=San Francisco; O=Heroku, Inc.; CN=*.herokuapp.com
*  start date: Jun 15 00:00:00 2020 GMT
*  expire date: Jul  7 12:00:00 2021 GMT
*  subjectAltName: host "ejava-docker.herokuapp.com" matched cert's "*.herokuapp.com"
*  issuer: C=US; O=DigiCert Inc; OU=www.digicert.com; CN=DigiCert SHA2 High Assurance
Server CA
*  SSL certificate verify ok.
> GET /api/hello?name=jim HTTP/1.1
> Host: ejava-docker.herokuapp.com
> User-Agent: curl/7.64.1
> Accept: */*
>
< HTTP/1.1 200
< Server: Cowboy
Hello, jim
```

Chapter 7. Summary

In this module we learned:

- to deploy an application under development to Heroku cloud provider to make it accessible to Internet users
    - using Docker form
- to deploy incremental and iterative changes to the application