codecentric Blog (https://blog.codecentric.de/)

IT knowledge from developers for developers

2021
POSTS

NEW (/EN/)

RSS-Feed (https://blog.codecentric.de/en/rss-feeds/)

AGILE (HTTPS://BLOG.CODECENTRIC.DE/EN/CATEGORY/AGILE-EN/)

ARCHITECTURE (HTTPS://BLOG.CODECENTRIC.DE/EN/CATEGORY/ARCHITECTURE/)

DATA (HTTPS://BLOG.CODECENTRIC.DE/EN/CATEGORY/DATA-EN/)

JAVA (HTTPS://BLOG.CODECENTRIC.DE/EN/CATEGORY/JAVA-EN/)

PERFORMANCE (HTTPS://BLOG.CODECENTRIC.DE/EN/CATEGORY/PERFORMANCE-EN/)

CONTINUOUS DELIVERY (HTTPS://BLOG.CODECENTRIC.DE/EN/CATEGORY/CONTINUOUS-DELIVERY-AGILE-EN/)

MICROSERVICES (HTTPS://BLOG.CODECENTRIC.DE/EN/CATEGORY/MICROSERVICES-ARCHITECTURE/)

CLOUD (HTTPS://BLOG.CODECENTRIC.DE/EN/CATEGORY/CLOUD-ARCHITECTURE/)

Overview (https://blog.codecentric.de/en/category/cicd/)

Core ML – inference on iOS (https://blog.codecentric.de/en/2019/08/core-ml-inference-on-ios/)

Spring Boot on Heroku with Docker, JDK 11 & Maven 3.5.x (https://blog.codecentric.de/en/2019/08/spring-boot-heroku-docker-jdk11/)

08/19/19 by **Jonas Hecht (https://blog.codecentric.de/en/author/jonas-hecht/)**

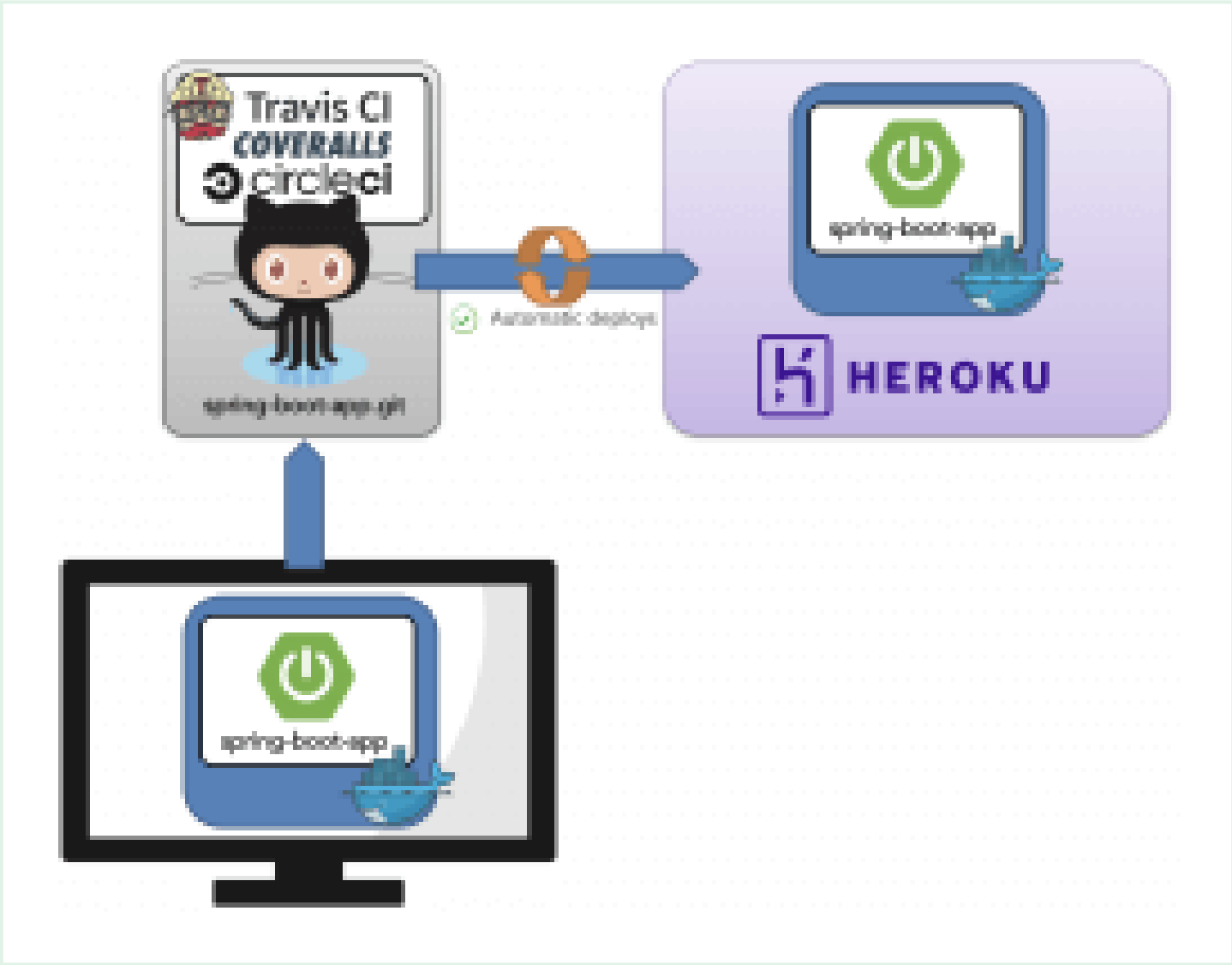
No Comments (https://blog.codecentric.de/en/2019/08/spring-boot-heroku-docker-jdk11/#comments)

If you don't know Heroku already, you'll get to love it soon! But wait – do you run Spring Boot apps based on JDK 11+? Do you build them with Maven 3.5.x? Maybe you should use Docker on Heroku – then this guide is for you!

Why I love Heroku

Using Heroku to run your apps is a really great experience! I fell in love with Heroku while running a charity project where we used a combination of hackathons and student support to build a holiday fun registration app for kids in my hometown. We first experimented with the tools the big cloud vendors provide themselves, but didn't have the time to invest what was needed to achieve a fully working software development process. We only had one day to set up a full continuous delivery pipeline including a working infrastructure and database until 20 students would come and wanted to build software.

But with Heroku this worked like a charm! As we used Java & Spring Boot in the backend, we could simply rely on the great getting-started guides about Java on Heroku . And there's also a great introductory article on how to deploy Spring Boot applications to Heroku (https://blog.codecentric.de/en/2015/10/deploying-spring-boot-applications-to-heroku/) by my former colleague Benedikt Ritter . To achieve a great development experience, just create a complementary GitHub repository for your application and the Heroku app itself. Also, be sure to connect your Heroku app to the new GitHub repo and activate the "Automatic deploys" feature, so that Heroku will build and deploy your application every time someone pushes into your GitHub project:



(https://blog.codecentric.de/files/2019/08/dev-process-heroku.png)

Logo sources: Docker logo , GitHub logo , TravisCI logo , Coveralls logo , CircleCI logo , Heroku logo , Spring Boot logo , Computer logo

With Heroku we had a great basis and delivered a registration app that is now running in beta test. Encouraged by this great experience, I used Heroku in many more of my Open Source projects.

Heroku's Java buildpack defaults to JDK 8...

So why am I writing this article? Well, lately I wanted to use Heroku for an example project that demonstrates the usage of a Spring Boot starter I am maintaining. The cxf-spring-boot-starter helps you get started extremely fast if you have to deal with good old SOAP webservices. Based on standard implementations for XML handling like JAX-B and JAX-WS, the starter uses the JDK-shipped modules `javax.xml.ws` and `javax.xml.bind`. But they were deprecated in Java 9 and removed from the JDK completely with Java 11. Using new dependecies, there is a way to overcome this problem. But as a consequence it's better to build the cxf-spring-boot-starter with JDK 11. If you develop a Spring Boot starter, it's often good practice to also provide some sample projects. This starter example project cxf-boot-simple is shipped and built with the starter – and therefore also needs a current JDK to for a successful build.

Since the default JDK on Heroku is currently version 8, we are slowly approaching the core problem that prompted me to write this article:

> “ Heroku currently uses OpenJDK 8 to run your application by default.

Configuring a newer JDK on Heroku is not a problem. According to the docs, we simply need to create a `system.properties` file inside the root of our application to configure this:

```
# Heroku configuration file
# see https://devcenter.heroku.com/articles/java-support#specifying-a-java-version
java.runtime.version=11
```

This should be it, right?! And it is! If you can live with that, you can stick to the standard Heroku Java buildpack and everything will be fine.

Heroku doesn't support Maven 3.5.x out of the box

But **if you need to build your software with a newer Maven version** also, you'll soon find yourself in the hell of being restricted to an old Maven version! Heroku currently only supports Maven versions <=3.3.9 . Using Heroku to build the Spring Boot starter project leads to the following error, which is related to Maven versions older than `3.5.x` (full stack trace here):

```
java.lang.IllegalArgumentException: Can not set org.eclipse.aether.spi.log.Logger field org.
```

Sadly we can't just simply use a newer Maven version in Heroku, although the `system.properties` file provides the appropriate configuration key `maven.version`. Versions newer than `3.3.9` are currently simply not supported. Also, the other escape route using the Maven Wrapper to use a newer Maven version didn't work in my case. Heroku simply ignored it and used Maven 3.3.9 again:

```
-----> Java app detected

-----> Installing JDK 11... done

-----> Installing Maven 3.3.9... done

-----> Executing: mvn -DskipTests clean dependency:list install

       [INFO] Scanning for projects...
```

Running Spring Boot apps with Docker on Heroku

Having no current Maven version available on Heroku made me think about a Java User Group Thüringen Talk by Kai Tödter about Spring Boot, REST & Angular , where he showed a demo of a Heroku deployment using Docker instead of a predefined buildpack. I remember discussing the pros and cons of using that additional layer on Heroku, since back then I wasn't convinced one really needs that.

But then I found myself in a situation where Docker would really help me out. Having Docker support, we could easily use the build tool in any specific version we wanted. And we would free ourselves from having to implement Heroku-specific configuration – if some day Heroku isn't the best choice, we can easily switch and run our Docker container somewhere else (I heard of somebody using that argument (https://blog.codecentric.de/en/2018/05/gitlab-ci-pipeline/) some time ago 😀). And the cool thing is: This described way of how to use Heroku is not restricted to Java – you can use it with nearly every language you have at hand!

And for sure there's also some curiosity involved – so let's just use Docker to run our Spring Boot apps on Heroku! There are two ways of how to use Docker on Heroku . The first is to use the Heroku Container Registry. It allows you to simply deploy your Docker images to Heroku. The

second option is to use Heroku to build our Docker images also for us, which we will use here. According to the docs , we only need a `Dockerfile` inside our sample project. Inside the project cxf-boot-simple the Dockerfile first looked like this:

```
# Docker multi-stage build

# 1. Building the App with Maven
FROM maven:3-jdk-11

ADD . /cxfbootsimple
WORKDIR /cxfbootsimple

# Just echo so we can see, if everything is there :)
RUN ls -l

# Run Maven build
RUN mvn clean install


# 2. Just using the build artifact and then removing the build-container
FROM openjdk:11-jdk

MAINTAINER Jonas Hecht

VOLUME /tmp

# Add Spring Boot app.jar to Container
COPY --from=0 "/cxfbootsimple/target/cxf-boot-simple-*-SNAPSHOT.jar" app.jar

# Fire up our Spring Boot app by default
CMD [ "sh", "-c", "java $JAVA_OPTS -Djava.security.egd=file:/dev/./urandom -jar /app.jar" ]
```

It puts a simple Docker multi-stage build into practice. The first build container uses a current Maven version and is therefore based on `maven:3-jdk-11`, which currently represents the latest tag of the Maven Docker image . The build artifact that results from a successful Maven build is

copied over as app.jar to the container running on Heroku later. As we don't want to mess with Maven-defined version numbers here, we simply use a `*` inside the path to the Spring Boot jar `cxf-boot-simple-*-SNAPSHOT.jar`.

Configuring Heroku to use Docker

Again the Heroku docs provide an excellent guide on how to configure and run your apps with Docker on Heroku. The key configuration file here is a heroku.yml inside the root of our project. This file provides us with four sections in which we can configure everything needed to build and run our apps with Docker. For example, if you need to have a Heroku addon running, you can configure that in the `setup` section. The build section is used to define the Docker build itself. The central part here is to tell Heroku where our `Dockerfile` resides:

```
build:
  docker:
    web: /cxf-spring-boot-starter-samples/cxf-boot-simple/Dockerfile
```

If you ever happen to do something between building and running your app with Docker, the third `release` phase comes to the rescue. Here you can provide CDNs with assets or run database schema migrations, for example.

The last section `run` defines the processes to run. If you've already run your apps without Docker on Heroku, you may know the `Procfile` used there to configure the startup behavior of your app. This file is ignored while using a `heroku.yml` – instead the run section will be used. But as you see inside the example project's heroku.yml, there's also another way (documented in the docs):

> If you do not include a run section in your heroku.yml manifest, the Dockerfile CMD is used instead.

And in our case where we want to use a Java backend inside our Docker container, the startup behavior should be always the same – be it on Heroku or on our local machine. Therefore I prefer to use the `CMD` keyword inside our Dockerfile!

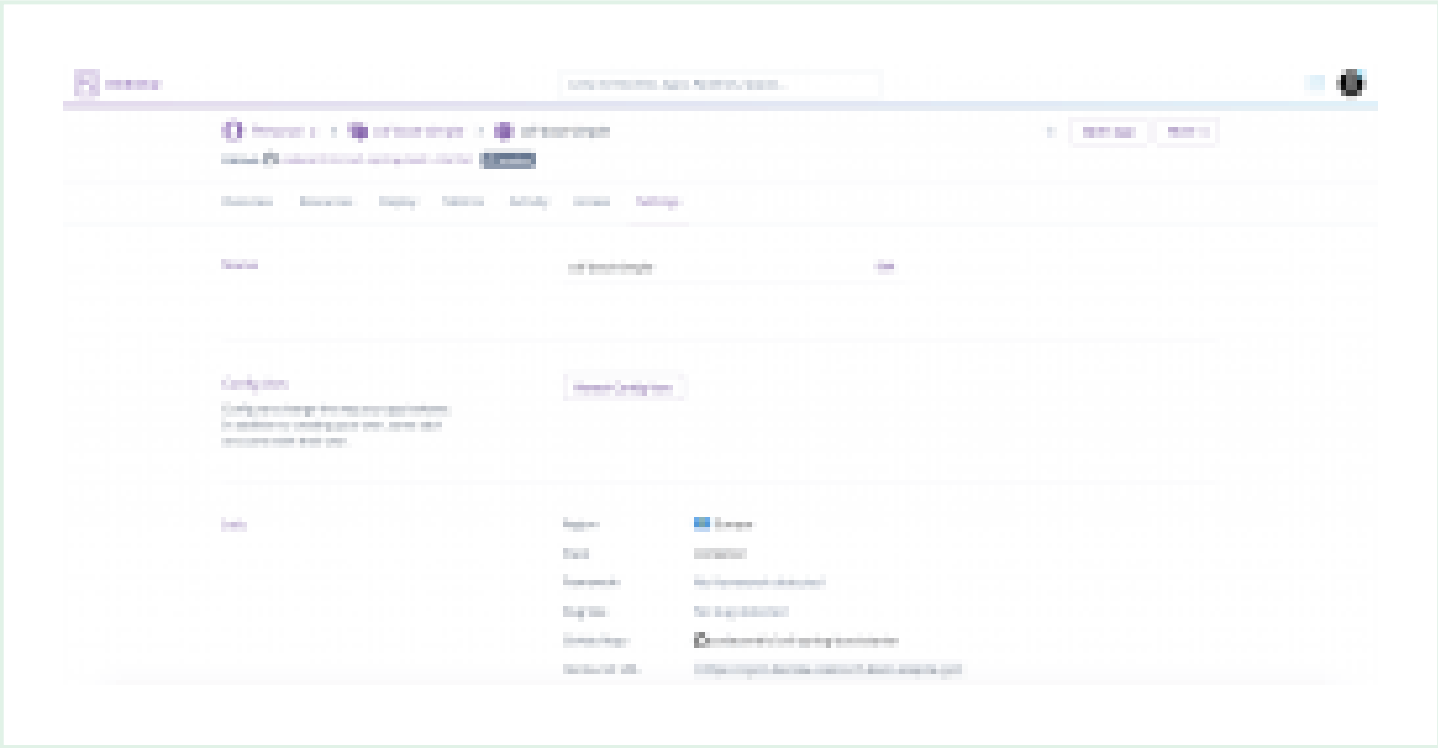
The last thing we need to configure for Heroku to use Docker instead is to change the Heroku stack of our used Dynos. The default `web` stack is preconfigured in Heroku. To change it, all you have to do is open up your commandline and execute the following:

```
heroku stack:set container
```

Just be sure to have the current heroku-cli installed – using your machine's package manager at best (like `brew install heroku`). If you have multiple apps running on Heroku, you maybe also need to define the concrete application with:

```
heroku stack:set container --app your-Heroku-app-name-here
```

The next push either to Heroku or to your connected GitHub repository should start your application inside a Docker container running on Heroku. Taking a look into the web console, you should also be able to find the newly configured `container` stack:



(https://blog.codecentric.de/files/2019/08/container-stack-application-heroku.png)

Preventing Error R14 (Memory quota exceeded)

This should be all you need to do. But wait. Aren't we running Java apps? Is our app really running on Heroku? Let's check the logs of our Heroku app using the Heroku CLI again:

```
$ heroku logs --tail --app your-Heroku-app-name-here
2019-07-24T02:58:48.253177+00:00 heroku[web.1]: Process running mem=836M(163.4%)
2019-07-24T02:58:48.253243+00:00 heroku[web.1]: Error R14 (Memory quota exceeded)
2019-07-24T02:58:55.236933+00:00 heroku[web.1]: State changed from starting to crashed
2019-07-24T02:58:55.111947+00:00 heroku[web.1]: Stopping process with SIGKILL
2019-07-24T02:58:55.217642+00:00 heroku[web.1]: Process exited with status 137
```

It seems like our app crashed because of a `Error R14 (Memory quota exceeded)` 🙁 Inside the guide about Troubleshooting Memory Issues in Java Applications there's also a section Configuring Java to run in a container, which describes how to configure the JVM correctly so it knows that it's running inside a container and should not reserve memory directly from the host machine. And as we use our Dockerfile's `CMD` to configure the JVM startup behavior, we need to extend that one instead of the `Procfile` (which is ignored using a `heroku.yml`):

```
# Fire up our Spring Boot app by default
CMD [ "sh", "-c", "java $JAVA_OPTS -XX:+UseContainerSupport -Djava.security.egd=file:/dev/./
```

Although the `-XX:+UseContainerSupport` option is the default from Java 10 on, I really like to set things explicitly if we rely on them. So let's leave this option set for JDK 10+ also.

The second thing we need to take care of is the correct JVM configuration for Heroku. The docs say:

> “ You'll see R14 errors in your application logs when this paging starts to happen.

But the docs also state that there should be defaults with correct `-Xmx` and `-Xss` settings provided out of the box by Heroku:

> The default support for most JVM-based languages sets -Xss512k and sets Xmx dynamically based on Dyno type. These defaults enable most applications to avoid R14 errors.

Digging deeper into the subject, I found that these defaults should be made transparent inside the `JAVA_OPTS` environment variable. But didn't we switch the Heroku default stack from `web` to `container`? Maybe we should take a look at the environment variables inside our Heroku Dyno to gain clarity. To do so, we can execute the command `printenv` with the help of Heroku CLI to see all environment variables inside:

```
$ heroku run printenv
Running printenv on ⬢ cxf-boot-simple... up, run.7988 (Free)
JAVA_URL_VERSION=11.0.4_11
HEROKU_EXEC_URL=https://exec-manager.heroku.com/a3ea58e6-d7b3-4fa8-8148-5567be41e46f
PORT=13303
JAVA_BASE_URL=https://github.com/AdoptOpenJDK/openjdk11-upstream-binaries/releases/download/
HOME=/
PS1=\[\033[01;34m\]\w\[\033[00m\] \[\033[01;32m\]$ \[\033[00m\]
JAVA_VERSION=11.0.4
TERM=xterm-256color
COLUMNS=160
PATH=/usr/local/openjdk-11/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
JAVA_OPTS=
LANG=C.UTF-8
JAVA_HOME=/usr/local/openjdk-11
PWD=/
LINES=32
DYNO=run.7988
```

And **here we go:** The `JAVA_OPTS` variable is simply empty! If you switched back to the default `web` stack configuration on Heroku, this variable would provide the right bits for us:

```
JAVA_OPTS=-Xmx300m -Xss512k -XX:CICompilerCount=2 -Dfile.encoding=UTF-8
```

So in order to prevent us from running into `Error R14 (Memory quota exceeded)` problem, we must be sure to tweak our application's Dockerfile:

```
# Fire up our Spring Boot app by default
CMD [ "sh", "-c", "java -Xmx300m -Xss512k -XX:CICompilerCount=2 -Dfile.encoding=UTF-8 -XX:+Us
```

Now our application should run on Heroku without any memory problems. If the error keeps occurring, there's also a good medium post on what can be done then.

Preventing Error R10 (Boot timeout)

We're nearly there! Another error might occur in the application's startup process, though:

```
2019-07-24T02:58:55.236933+00:00 heroku[web.1]: State changed from starting to crashed
2019-07-24T02:58:55.111947+00:00 heroku[web.1]: Error R10 (Boot timeout) -> Web process fail
2019-07-24T02:58:55.111947+00:00 heroku[web.1]: Stopping process with SIGKILL
2019-07-24T02:58:55.217642+00:00 heroku[web.1]: Process exited with status 137
```

Did we set the `$PORT` environment variable correctly? Let's look into a `Procfile` which we needed to use in the pre-Docker era on Heroku. It also had to contain the `$PORT` variable so that Spring Boot is able to launch its internal Tomcat accordingly:

```
web: java -Dserver.port=$PORT -jar cxf-spring-boot-starter-samples/cxf-boot-simple/target/cx
```

And for sure this configuration is also needed inside our `Dockerfile`! Because the docs state :

> The web process must listen for HTTP traffic on `$PORT`, which is set by Heroku. `EXPOSE` in Dockerfile is not respected, but can be used for local testing. Only HTTP requests are supported.

So let's tweak our example project's Dockerfile again:

```
# Fire up our Spring Boot app by default
CMD [ "sh", "-c", "java -Dserver.port=$PORT -Xmx300m -Xss512k -XX:CICompilerCount=2 -Dfile.e
```

Now the `$PORT` environment variable should be used to fire up our Spring Boot app. To verify this, we can execute our Docker container locally. Here we also see another advantage of using Docker with Heroku: we can simply test things locally, which drastically reduces the time we have to invest into our development process.

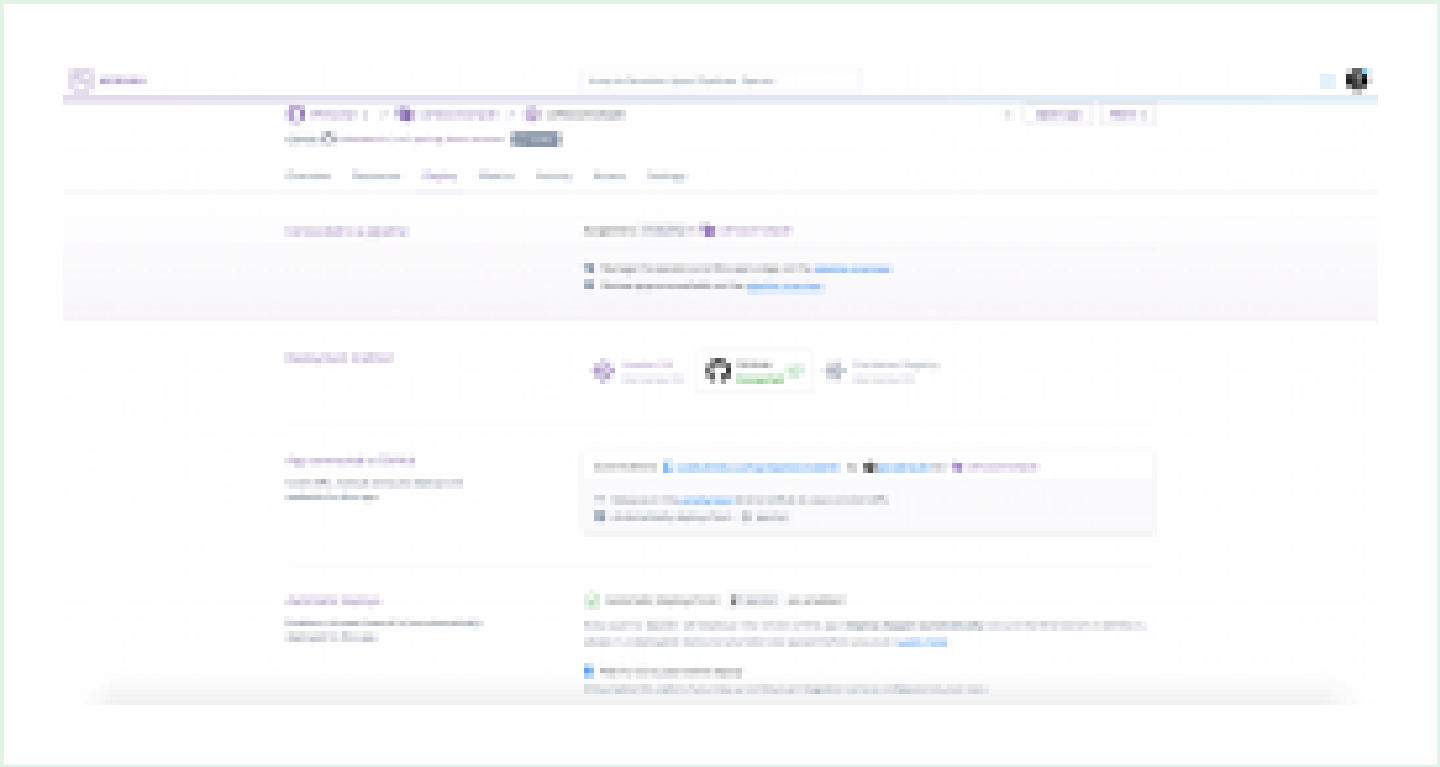
Now just be sure to append `PORT` as environment variable in the `docker run` command:

```
docker build . --tag cxfbootsimple
docker run -e "PORT=8095" cxfbootsimple
```

Our application should be running now. So we can go on and take a look into our container. Simply use `docker ps` to get the running container's ID and then open up a bash inside it:

```
docker exec -it containerId bash
curl localhost:8095/my-foo-api -v
```

If the curl command outputs some HTML page, it should be good enough to push our updated Dockerfile into our application's GitHub repository. Having connected Heroku to our repository and configured it to do automatic deploys, the push should result in a new Heroku deployment:



(https://blog.codecentric.de/files/2019/08/heroku-automatic-deploys.png)

Finally, our Spring Boot app should be running successfully with Docker on Heroku! You might want to check out this article's example project cxf-boot-simple , where you can find the running Heroku app at https://cxf-boot-simple.herokuapp.com/my-foo-api .

Spring Boot on Heroku with Docker

I'am absolutely relieved I can keep using my beloved Heroku for that use case also! The simplicity of this development process is impressive – as a DevOps fanboy I know what I'm talking about. Using Docker, we combine the simplicity of Heroku with the power of Docker. Just remember: now we use mostly the same infrastructure both locally and in the cloud! And we're not bound to restrictions of some predefined Heroku buildpacks – not even to a specific programming language! The same process described here could be used for any project – simply change your Dockerfile accordingly and you're done. Have fun with Docker on Heroku! 🙂

Tags

CLOUD (HTTPS://BLOG.CODECENTRIC.DE/EN/TAG/CLOUD-EN/) DOCKER (HTTPS://BLOG.CODECENTRIC.DE/EN/TAG/DOCKER/)

HEROKU (HTTPS://BLOG.CODECENTRIC.DE/EN/TAG/HEROKU/) JAVA11 (HTTPS://BLOG.CODECENTRIC.DE/EN/TAG/JAVA11/)

SPRING BOOT (HTTPS://BLOG.CODECENTRIC.DE/EN/TAG/SPRING-BOOT-EN-2/)

Jonas Hecht (https://blog.codecentric.de/en/author/jonas-hecht/)

Trying to bridge the gap between software architecture and hands on coding, Jonas hired at codecentric. He has deep knowledge in all kinds of enterprise software development, paired with passion for new technology. Connecting systems via integration frameworks Jonas learned to not only get the hang of technical challenges.

Post by **Jonas Hecht**

CONTINUOUS DELIVERY

Simplifying Spring Boot GraalVM Native Image builds with the native-image-maven-plugin (https://blog.codecentric.de/en/2020/06/spring-boot-graalvm-native-image-maven-plugin/)

CONTINUOUS DELIVERY

Running Spring Boot GraalVM Native Images with Docker & Heroku (https://blog.codecentric.de/en/2020/06/spring-boot-graalvm-docker-heroku/)

More content about **CICD**

CICD

Remote training with GitLab-CI and DVC (https://blog.codecentric.de/en/2020/01/remote-training-gitlab-ci-dvc/)

CICD

AWS CDK Part 6: Lessons learned (https://blog.codecentric.de/en/2019/11/aws-cdk-part-6-lessons-learned/)

Comment

Nachricht

Name

E-Mail

Save my name, email, and website in this browser for the next time I comment.

SUBMIT