A Quick Guide to Deploying Java Apps on OpenShift

by Piotr Mińkowski MVB · May. 20, 18 · Java Zone · Tutorial

In this article, I'm going to show you how to deploy your applications on OpenShift (**Minishift**), connect them with other services exposed there, or use some other interesting deployment features provided by OpenShift. OpenShift is built on top of Docker containers and the Kubernetes container cluster orchestrator.

Running Minishift

We use Minishift to run a single-node OpenShift cluster on the local machine. The only requirement before installing MiniShift is having a virtualization tool installed. I use Oracle VirtualBox as a hypervisor, so I should set the `--vm-driver` parameter to `virtualbox` in my running command.

```
$ minishift start --vm-driver=virtualbox --memory=3G
```


Running Docker

It turns out that you can easily reuse the Docker daemon mana rder to run Docker commands directly from your command line without any additional installation. To achieve this, just run the following command after starting Minishift.

```
@FOR /f "tokens=* delims=^L" %i IN ('minishift docker-env') DO @call %i
```


Running the OpenShift CLI

The last tool that is required before starting any practical e . It is available under the command `oc`. To enable it on your command line, run the following

available under the command `oc`. To enable it on your command line, run the following commands:

```
$ minishift oc-env
$ SET PATH=C:\Users\minkowp\.minishift\cache\oc\v3.9.0\windows;%PATH%
$ REM @FOR /f "tokens=*" %i IN ('minishift oc-env') DO @call %i
```

Alternatively, you can use the OpenShift web console which is available under port **8443**. On my Windows machine, it is, by default, launched under the address **192.168.99.100**.

Building Docker Images of the Sample Applications

I prepared the two sample applications that are used for the purposes of presenting OpenShift deployment process. These are simple Java and Vert.x applications that provide an HTTP API and store data in MongoDB. We need to build Docker images with these applications. The source code is available on GitHub in the branch openshift. Here's a sample Dockerfile for `account-vertx-service`.

```
FROM openjdk:8-jre-alpine
ENV VERTICLE_FILE account-vertx-service-1.0-SNAPSHOT.jar
ENV VERTICLE_HOME /usr/verticles
ENV DATABASE_USER mongo
ENV DATABASE_PASSWORD mongo
ENV DATABASE_NAME db
EXPOSE 8095
COPY target/$VERTICLE_FILE $VERTICLE_HOME/
WORKDIR $VERTICLE_HOME
ENTRYPOINT ["sh", "-c"]
CMD ["exec java -jar $VERTICLE_FILE"]
```


Go to the `account-vertx-service` directory and run the follov ild an image from the Dockerfile visible above.

```
$ docker build -t piomin/account-vertx-service .
```

The same steps should be performed for `customer-vertx-service`. After you will have two images built, both in the same version `latest`, which now can be deployed and run on Minishift.

Preparing the OpenShift Deployment Descriptor

When working with OpenShift, the first step of our application's deployment is to create a YAML configuration file. This file contains basic information about the deployment like the containers used for running applications **(1)**, scaling **(2)**, triggers that drive automated deployments in response to events **(3)**, or a strategy of deploying your pods on the platform **(4)**.

```
kind: "DeploymentConfig"
apiVersion: "v1"
metadata:
  name: "account-service"
spec:
  template:
    metadata:
      labels:
        name: "account-service"
    spec:
      containers: # (1)
        - name: "account-vertx-service"
          image: "piomin/account-vertx-service:latest"
          ports:
            - containerPort: 8095
              protocol: "TCP"
  replicas: 1 # (2)
  triggers: # (3)
    - type: "ConfigChange"
    - type: "ImageChange"
      imageChangeParams:
        automatic: true
        containerNames:
          - "account-vertx-service"
        from:
          kind: "ImageStreamTag"
          name: "account-vertx-service:latest"
  strategy: # (4)
    type: "Rolling"
  paused: false
  revisionHistoryLimit: 2
```


Deployment configurations can be managed with the `oc` command like any other resource. You can create a new configuration or update the existing command.

```
$ oc apply -f account-deployment.yaml
```

You might be a little surprised, but this command does not trigger any build and does not start the pods. In fact, you have only created a resource of type `deploymentConfig`, which describes the deployment process. You can start this process using some other `oc` commands, but first, let's take a closer look at the resources required by our application.

Injecting Environment Variables

As I have mentioned before, our sample applications use an external datasource. They need to open the connection to the existing MongoDB instance in order to store their data passed using HTTP endpoints exposed by the application. Here's our `MongoVerticle` class, which is responsible for establishing a client connection with MongoDB. It uses environment variables for setting security credentials and a database name.

```
public class MongoVerticle extends AbstractVerticle {

    @Override
    public void start() throws Exception {
        ConfigStoreOptions envStore = new ConfigStoreOptions()
                .setType("env")
                .setConfig(new JsonObject().put("keys", new JsonArray().add("DATABASE_USER").a
        ConfigRetrieverOptions options = new ConfigRetrieverOptions().addStore(envStore);
        ConfigRetriever retriever = ConfigRetriever.create(vertx, options);
        retriever.getConfig(r -> {
            String user = r.result().getString("DATABASE_USER");
            String password = r.result().getString("DATABASE_PASSWORD");
            String db = r.result().getString("DATABASE_NAME");
            JsonObject config = new JsonObject();
            config.put("connection_string", "mongodb://" + user +                    godb/"
            final MongoClient client = MongoClient.createShared(v
            final AccountRepository service = new AccountRepositoryImpl(client);
            ProxyHelper.registerService(AccountRepository.class, vertx, service, "account-serv
        });
    }

}
```


MongoDB is available in OpenShift's catalog of predefined Docker images. You can easily

MongoDB is available in OpenShift's catalog of predefined Docker images. You can easily deploy it on your Minishift instance just by clicking the *"MongoDB"* icon in *"Catalog"* tab. Your username and password will be automatically generated if you do not provide them during the deployment setup. All the properties are available as deployment environment variables and are stored as `secrets/mongodb`, where `mongodb` is the name of the deployment.

Environment variables can be easily injected into any other deployments using the `oc set` command, and therefore, they are injected into the pod after performing the deployment process. The following command injects all secrets assigned to the `mongodb` deployment to the configuration of our sample application's deployment.

```
$ oc set env --from=secrets/mongodb dc/account-service
```

Importing Docker Images to OpenShift

A deployment configuration is ready. So, in theory, we could have started the deployment process. However, let's go back for a moment to the deployment config defined in **Step 5**, the section on deployment descriptors. We defined two triggers that cause a new replication controller to be created, which results in deploying a new version of the pod. The first of them is a configuration change trigger that fires whenever changes are detected in the pod template of the deployment configuration (`ConfigChange`).

The second of them, the image change trigger (`ImageChange`), fires when a new version of the Docker image is pushed to the repository. To be able to see whether an image in a repository has been changed, we have to define and create an image stream. An image stream does not contain any image data, but presents a single virtual view of something similar to an image repository. Inside the deployment config file we have defined an image stream `account-vertx-service`, so the same name should be provided inside the image stream definition. In turn, when setting the `spec.dockerImageRepository` field, we define the Docker pull specification for the image.

```
apiVersion: "v1"
kind: "ImageStream"
metadata:
  name: "account-vertx-service"
spec:
  dockerImageRepository: "piomin/account-vertx-service"
```

Finally, we can create resource on OpenShift platform.

```
$ oc apply -f account-image.yaml
```

Running the Deployment

Once a deployment configuration has been prepared, and the Docker images have been successfully imported into the repository managed by the OpenShift instance, we may trigger the build using the following `oc` commands.

```
$ oc rollout latest dc/account-service
$ oc rollout latest dc/customer-service
```

If everything goes fine, the new pods should be started for the defined deployments. You can easily check it out using the OpenShift web console.

Updating the Image Streams

We have already created two image streams related to the Docker repositories. Here's the screen from the OpenShift web console that shows the list of available image streams.

Name	Docker Repo	Tags	Updated
customer-vertx-service	172.30.1.1:5000/sample-deployment/customer-vertx-service	latest	7 minutes ago
account-vertx-service	172.30.1.1:5000/sample-deployment/account-vertx-service	latest	22 minutes ago

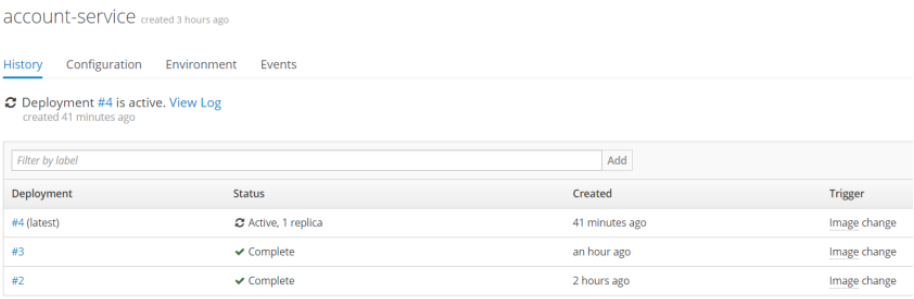
To be able to push a new version of an image to OpenShift's internal Docker registry, we should first perform a `docker login` against this registry using the user's authentication token. To obtain the token from OpenShift, use the `oc whoami` command and pass it to your `docker login` command with the `-p` parameter.

```
$ oc whoami -t
Sz9_TXJQ2nyl4fYogR6freb3b0DGlJ133DVZx7-vMFM
$ docker login -u developer -p Sz9_TXJQ2nyl4fYogR6freb3b0DGlJ133DVZx7-vMFM https://172.30.1.1:
```

Now, if you perform any change in your application and rebuild your Docker image with the `latest` tag, you have to push that image to the image stream on OpenShift. The address of the internal registry has been automatically generated by OpenShift and you can find out in the image stream's details. For me, it is **172.30.1.1:5000**.

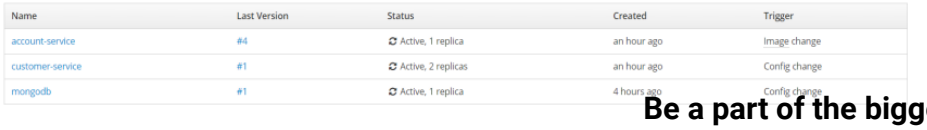
```
$ docker tag piomin/account-vertx-service 172.30.1.1:5000/sample-deployment/account-vertx-serv
$ docker push 172.30.1.1:5000/sample-deployment/account-vertx-service
```

After pushingthe new version of the Docker image to the image stream, a rollout of the application is started automatically. Here's the screen from the OpenShift web console that shows the history of **account-service**'s deployments.



Conclusion

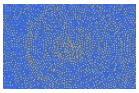
I have shown you the steps of deploying your application on the OpenShift platform. Based on a sample Java application that connects to a database, I illustrated how to inject credentials to that application's pod entirely transparently for a developer. I also perform an update of the application's Docker image in order to show how to trigger a new deployment upon image change.



Like This Article? Read More From DZone

Deploying Docker Images to OpenShift

Container Images for OpenShift (Part 2): Structuring Images



Docker Swarm Mode on AWS

Free DZone Refcard
Getting Started With Headless CMS

Topics: JAVA , OPENSHIFT , DOCKER CONTAINERS , CONTAINER ORCHESTRAT