Java Containerization:

Containerizing Your Java Applications

ORIGINAL BY AMJAD AFANAH, CEO AT DCHQ, INC.
WRITTEN BY RAFAEL BENEVIDES, DIRECTOR OF DEVELOPER EXPERIENCE AT REDHAT

CONTENTS

WHAT IS A LINUX CONTAINER?

A Linux container is an operating system-level virtualization technology that, unlike a virtual machine (VM), shares the host operating system kernel and makes use of the guest operating system libraries for providing required OS capabilities. Since there is no dedicated operating system, containers are more lightweight and start much faster than VMs.

The history of Linux containers can be traced back to `chroot`, which was introduced in 1979 as a UNIX operating-system system call that provides an isolated disk space for each process. Later, in 1982, this was added to BSD. More technologies have emerged since then, including FreeBSD Jails, Linux VServer, Solaris Containers, OpenVZ, Process Containers, Control Groups, LXC, Warden, LMCTFY, and others.

In 2013, Docker was developed as an open platform for packaging, deploying, and running distributed applications. Docker uses its own Linux container library called libcontainer. It became the most popular and widely used container management system. Later, this library was donated to OCI (Open Container Initiative) with the name of runc.

This Refcard will focus on the design, deployment, service discovery, and management of Java applications on the open-source project called Docker.

DOCKER'S ARCHITECTURE

Docker uses a typical client-server architecture. The Docker client talks to the Docker daemon, which does the heavy lifting of building, running, and distributing Docker containers. The Docker client and daemon communicate via sockets or through a RESTful API.

REFERENCE	DESCRIPTION
Docker Images	Read-only templates that use union file systems to combine layers — making them very lightweigh. Images are build from Docker files.
Docker Registries	Store Docker images. Users can push (or publish) their images to a public registry (like Docker Hub) or their own registry behind a firewill. resistries store the "tagged" images — allowing users to maintain different versions of the same image.

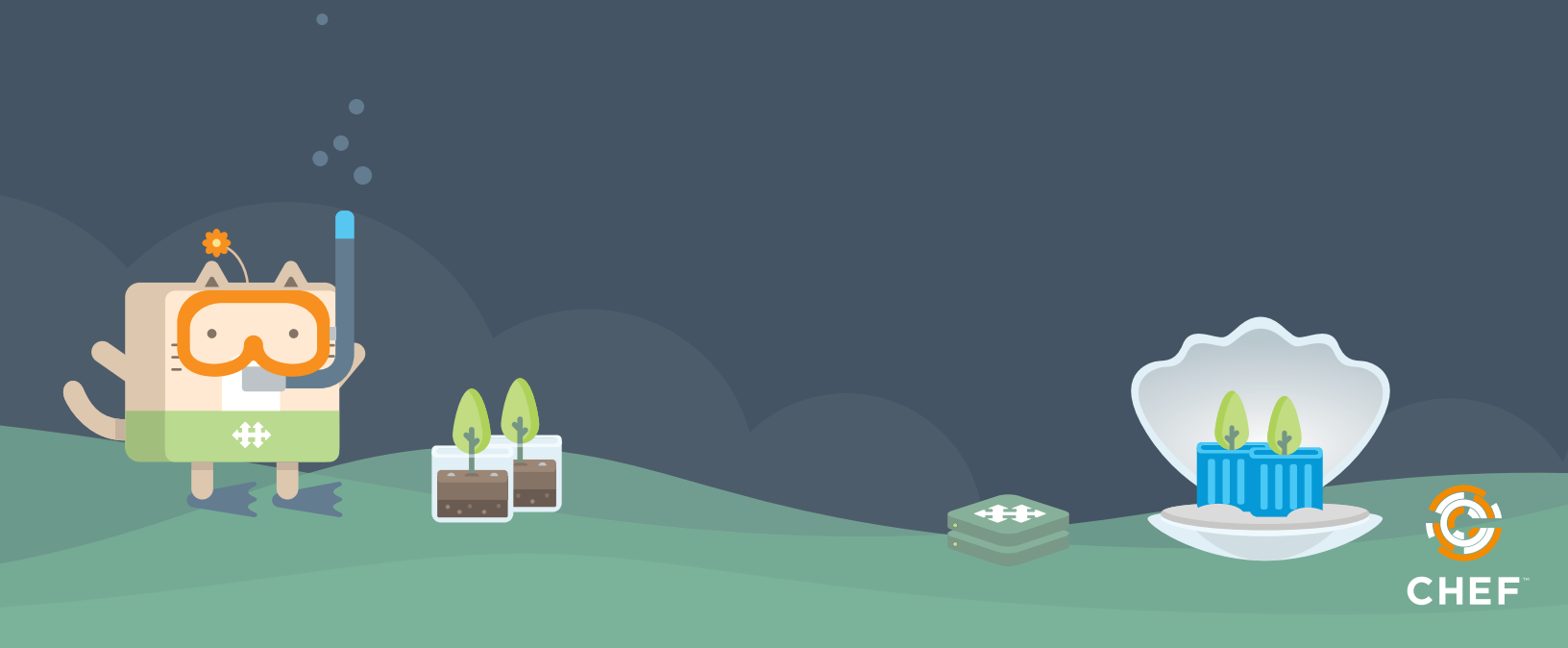
habitat

Get the most of your Node.JS apps with Habitat

TARGET ANY CLOUD, ANY CONTAINER, WITH A SINGLE BUILD.

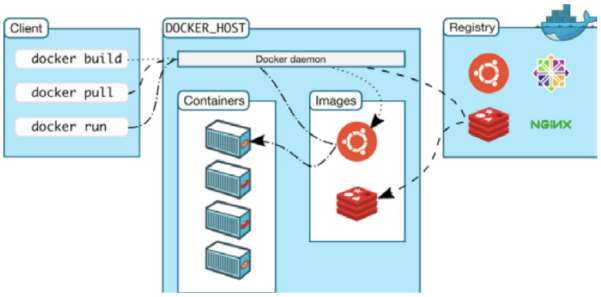
Habitat from Chef automates the build and packaging of Node.JS apps so you can target any cloud, or any container runtime from a single build output. Connect your GitHub repo to Habitat, check in code, and automatically create a build artifact that can run on AWS, Azure, GCP, using Docker, Kubernetes and more. And the artifact is automatically published with management code for DevOps bliss.

Get started in 10 minutes at http://habitat.sh



CHEF

Docker Containers	Virtualized application environments that run on a Docker Host in isolation. Containers are launched from Docker images, adding a read-write layer on top of the image (using a union file system) as well as the network/bridge interface and IP address. When a container is launched, the process specified in the Dockerfile is executed and the logs are captured for auditing and diagnostics.
Dockerfiles	Composed of various commands (instructions) listed successively to automatically perform actions on a base image in order to create a new one. The instructions specify the operating system, application artifacts, data volumes, and exposed ports to be used, as well as the command (or script) to run when launching a Docker container.
Docker Host	A Linux host (either a physical/bare-metal server or a virtual machine) that is running a Docker daemon on which images can be built, pulled or pushed and containers can run in isolation.
Docker Client	Command-line utility or other tool that takes advantage of the Docker API (docs.docker.com/reference/api/docker_remote_api) to communicate with a Docker daemon



HOW IS DOCKER DIFFERENT THAN JVM?

The JVM is Java's solution for application portability across different platforms — but Docker provides a different kind of virtualization that makes use of the guest operating system libraries, not just the Java application. When a Docker container is launched, a filesystem is allocated, along with the network/bridge interface and IP address. The command (or script) specified in the Dockerfile used to build the underlying Docker image is then executed, and the resulting Linux process runs in isolation.

As a result, Docker can be used to package an entire JVM along with the JAR or WAR files and other parts of the application into a single container that can run on any Linux host consistently. This eliminates some of the challenges associated with making sure that the right JAR file version is used on the right JVM. Moreover, CPU and memory resource controls can be used with Docker containers — allowing users to allocate maximum amounts of resources that an application can use.

DOCKER'S BENEFITS

The main advantages of Docker are:

- **Application portability**: Docker containers run exactly the same on any Linux host. This eliminates the challenge of deploying applications across different compute infrastructure (e.g. local developer machine, VM, or in the cloud).
- **Higher server density**: The lightweight nature of containers allows developers to optimize server utilization for their application workloads by running more containers on the same host while achieving the same isolation and resource allocation benefits of virtual machines.

DOCKER CE VS. EE

Docker, Inc. provides two versions of its Docker project. Docker CE (Community Edition) is meant for developers who are experimenting with containers. They also provide the EE (Enterprise Edition), which is the supported and certified version.

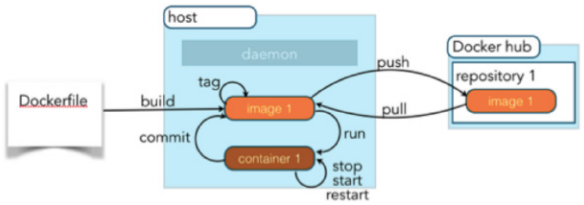
Because Docker EE requires a subscription, this Refcard will use the Community Edition.

DOCKER BASIC WORKFLOW

The typical workflow in Docker involves building an image from a Dockerfile or pulling an image from a registry (like Docker Hub). Once the image is available on the Docker Host, a container can be launched as a runtime environment. Docker Hub has approximately 100 "official" images published by software vendors — eliminating the need to build a Tomcat or MySQL image from scratch in most cases. Once a container is running, it can be stopped, started, or restarted using the CLI. If changes are made to the container, a user can commit the

changes made into a new image with either the same tag (or version) or a different one. The new image can, of course, then be pushed to a registry (like Docker Hub).

Instead of listing all the supported workflows, this Refcard walks through a basic Spring Boot Java application — starting from basic container's use cases to more advanced ones.



SETTING UP DOCKER

INSTALLING DOCKER MANUALLY

You can refer to Docker's official documentation for detailed installation instructions. For Fedora Linux, for example, please refer to this document.

You can use this simple command to install Docker CE:

```
$ sudo dnf install docker-ce
```

Make sure the Docker daemon has been started and is running:

```
$ sudo systemctl start docker
```

DOCKER HELLO WORLD

Now that you have the Docker daemon running, you can run your first container by typing:

```
$ sudo docker run hello-world
Unable to find image
\'hello-world:latest\' locally
latest: Pulling from library/hello-world
9db2ca6ccae0: Already exists
Digest:sha256:4b8ff392a12ed9ea17784bd3c9a8b1fa3299c
ac44aca35a85c90c5e3c7afacdc
Hello from Docker!
. . .
```

Note that since the image *hello-world* doesn't exist yet on the Docker daemon that's running on your machine, it will pull the image from Docker Hub. This process is similar to what happens with Maven, which will download the Maven dependencies from Maven Central and store them on your local repository. After downloading the image, a container will be created and executed.

DOCKER HELLO WORLD WITH JAVA

Although there's an official Docker image for OpenJDK, it doesn't provide any application in the image. The OpenJDK image only contains the JDK to allow the execution of Java applications.

However, companies and developers can create their own non-official Docker image and store it at Docker Hub. This is the case for the JBoss WildFly Application Server. Let's try this image:

```
 $ sudo docker run -it jboss/wildfly
=========================================================
  JBoss Bootstrap Environment
  JBOSS_HOME: /opt/jboss/wildfly
  JAVA: /usr/lib/jvm/java/bin/java
  JAVA_OPTS:  -server -Xms64m -Xmx512m
-XX:MetaspaceSize=96M -XX:MaxMetaspaceSize=256m -Djava.
net.preferIPv4Stack=true -Djboss.modules.system.
pkgs=org.jboss.byteman -Djava.awt.headless=true
=========================================================
. . .
Full 12.0.0.Final (WildFly Core 4.0.0.Final) started in
4144ms - Started 292 of 513 services (308 services are
lazy, passive or on-demand)
```

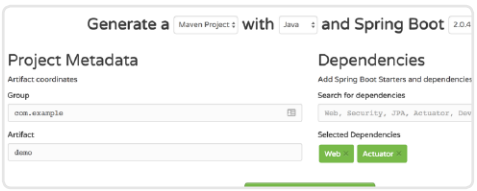
In the command above, *jboss* is the username, while *wildfly* is the image name provided by that user. Note that the switch `-i` enables interactivity and `-t` assigns a terminal.

It's also possible to specify tags for the version. In that case, for example, if you want to execute a specific version of WildFly, you can add the tag to the command. In the following example, we used `9.0.0.Final` as the tag.

```
 $ sudo docker run -it jboss/wildfly:9.0.0.Final
. . .
23:25:10,745 INFO  [org.jboss.as] (Controller Boot
Thread) WFLYSRV0025: WildFly Full 9.0.0.Final (WildFly
Core 1.0.0.Final) started in 2612ms - Started 203 of
379 services (210 services are lazy, passive or
on-demand)
```

CONTAINERIZING A SPRING BOOT DEMO APPLICATON

Go to https://start.spring.io/ and create a Maven application. Select **Web** and **Actuator** as dependencies. Click **Generate Project**, download the file *demo.zip*, and extract it.



For test purposes, let's add a simple REST endpoint that returns a message with the hostname and the number of invocations.

Add the following class to your application. You can type the code or copy from here.

```
package com.example.demo;
import org.springframework.web.bind.annotation.
GetMapping;
import org.springframework.web.bind.annotation.
RequestMapping;
import org.springframework.web.bind.annotation.
ResponseBody;
import org.springframework.web.bind.annotation.
RestController;
@RestController
@RequestMapping("/api/hello")
public class HelloController {
    private static final String RESPONSE_STRING_FORMAT
= "Hello from '%s': %d\n";
    /**
     * Counter to help us see the lifecycle
     */
    private int count = 0;
    @GetMapping
    public String helloworld(){
        count++;
        return String.format(
                RESPONSE_STRING_FORMAT,
                System.getenv().
getOrDefault("HOSTNAME", "unknown"),
                count);
    }
}
```

Now, go to your application folder and execute the following Maven command to start your Spring Boot application. This will allow you to test the application before placing it inside a container.

```
$ mvn spring-boot:run
[INFO] Scanning for projects...
. . .
INFO 2172 --- [           main] com.example.demo.
DemoApplication: Started DemoApplication in 2.614
seconds (JVM running for 5.447)
```

Open the browser in the following URL: http://localhost:8080/api/hello.

Note that every access (or refresh) will increase the counter variable.

Before moving to the next section, make sure to stop the application running locally through CTRL + C.

CREATING A CONTAINER IMAGE FOR THE SPRING BOOT APPLICATION

The easiest way to create a Docker image is to write a Dockerfile. It contains a series of commands that customize an image based on a previous one. For our application, we will start from the existing OpenJDK 10 image, add our JAR file, and define the default command that will execute when the container starts.

The Dockerfile below is also available to be copied here. This file should be in the same folder of the project.

```
# Our base image that contains OpenJDK
FROM openjdk

# Add the fatjar in the image
COPY target/demo-0.0.1-SNAPSHOT.jar /

# Default command
CMD java -jar /demo-0.0.1-SNAPSHOT.jar
```

Before adding the JAR file to the image, we needed to create it using the command `mvn package`. Now that we have a Dockerfile, we can execute the build of the JAR and the Docker image using the following commands:

```
$ mvn package
. . .

$ sudo docker build -t demo .
Sending build context to Docker daemon  17.87MB
Step 1/3 : FROM openjdk
Step 2/3 : COPY target/demo-0.0.1-SNAPSHOT.jar /
Step 3/3 : CMD java -jar /demo-0.0.1-SNAPSHOT.jar
. . .
Successfully built 4df92f5aa7f6
Successfully tagged demo:latest
```

The flag `-t` on the `docker build` command specifies the name/tag of the image. Using a tag called `demo` allows you to refer to this image by its name when you need to create a container instance. The last parameter for `docker build` is the path of Dockerfile. Since the Dockerfile is in the same folder of your project, the `.` will instruct the Docker daemon to use the file in the same folder that you execute the command.

RUNNING THE APPLICATION INSIDE THE CONTAINER

The previously created image called *demo* is available locally. You can check the existing images available locally through the command `docker images`.

The *demo* image can now be used to create a new container instance with the following command:

```
$ sudo docker run -d --name demo-app -p 8080:8080 demo
582b891f7ffa307ad08f6669cfb473ac822dc49d29b80dc18477d4
a120d2a023
```

The flag `-d` instructs the Docker daemon to detach from the container after the execution of the command. The flag `--name` gives the name *demo-app* to this container. If you don't specify a name, Docker will create a random name for you.

By default, you can't access any ports in the container unless you specify the mapping between the Docker daemon port and the container port. This feature is useful to allow the same application that is running on port 8080 to bind to different ports if you decide to run multiple containers. In our case, we are mapping port 8080 from the Docker daemon to port 8080 of the container through the flag `-p 8080:8080`.

Finally, we specify the name of the image that we want to use to create the container. In this case, we used the *demo* image created previously. After the execution of the command, it will show a hash number that identifies your container. You can verify the containers that are running with the following command:

```
$ sudo docker ps
CONTAINER ID    IMAGE    COMMAND    CREATED .   STATUS
PORTS           NAMES
ebc74b33f1e5    demo  "/bin/sh -c 'java -j…"   5
minutes ago       Up 5 minutes
0.0.0.0:8080->8080/tcp   demo-app
```

Now that your container is being executed, you can open your browser again in the URL http://localhost:8080/api/hello. You will see the same result that you had previously, but this time, your application is running inside the container.

EXTRA OPERATIONS ON THE CONTAINER

Although your container is running detached, you can perform some operations on it like checking the logs of your application with the command `docker logs` plus the name of the container.

```
$ sudo docker logs demo-app

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v2.0.4.RELEASE)
...
```

If you want to open a terminal session inside the container, you can use the command `docker exec` followed by the name of the container and the name of a process that you want to execute. In this case, we want to execute `bash` in an interactive terminal. Use the following command:

```
$ sudo docker exec -it demo-app bash
root@ebc74b33f1e5:/# ls
bin  dev  home   lib64   mnt   root srv  usr
boot  docker-java-home  lib    libx32  optrun sys  var
demo-0.0.1-SNAPSHOT.jar etc  lib32 media proc sbin tmp
```

Once you are inside the container, you can run any Linux command like `ls` or `ps`. Note that if you run `ls`, you should see the file *demo-0.0.1-SNAPSHOT.jar* that was added during the creation of the image. You can exit from the container terminal and return to your local terminal by typing `exit`.

PUBLISHING YOUR IMAGE ON DOCKER HUB

To publish your image on Docker Hub, you need to sign up for a free account.

Now that you have a Docker Hub account, you can log in by running `docker login`:

```
$ sudo docker login
Login with your Docker ID to push and pull images from
Docker Hub.If you don't have a Docker ID, head over to
https://hub.docker.com to create one.
Username:
Password:
Login Succeeded
```

Note that the images published on Docker Hub need to have your username as a prefix. When we created our *demo* image, we didn't add the username prefix, but we can do it now using the command `docker tag`.

```
$ docker tag demo <your username>/refcard-demo
```

Now, the same image has two tags (*demo* and *rafabene/refcard-demo*). You can use any tag that you want. The only requirement to push to Docker Hub is to have your username as a prefix.

To push the new tag to Docker Hub, just execute the command `docker push \<your username\>/image-name`. Example:

```
$ sudo docker push <your username>/refcard-demo
The push refers to repository [docker.io/<yourusername>
/refcard-demo]
4c7628270fc7: Pushed
. . .
25edbec0eaea: Mounted from library/openjdk
latest: digest:
```

CODE BLOCK CONTINUED ON FOLLOWING PAGE

```
sha256:92cddb652d55db14c3f697d25f4c93a145f
9b287522ae43afe623fe130437d35 size: 2212
```

Now, your image */refcard-demo* is available to be used by any Docker daemon worldwide.

SCALING YOUR APPLICATION IN THE CLOUD

Every time you execute the `docker run` command, it creates a single container running in a single machine. Now, imagine that you need to execute a fleet of containers on multiple servers. You will face the following challenges:

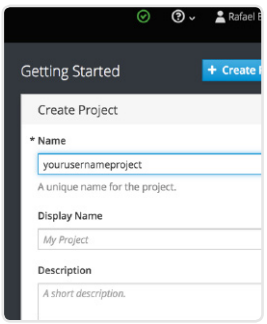
- How to scale?
- How to avoid port conflicts?
- How to manage them in multiple hosts?
- What happens if a host has a trouble?
- How to keep them running?
- How to update them?
- Where are my containers?

The answer to address all these challenges is using a container orchestration solution like Kubernetes. The easiest way to try Kubernetes and also have other features like image build automation, deploy automation, self-service catalog, and CI/CD pipelines is through the usage of OpenShift.

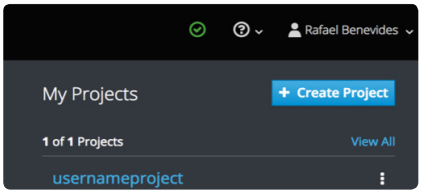
Red Hat® OpenShift® is the industry's most secure and comprehensive enterprise-grade container platform based on industry standards, Docker, and Kubernetes. It is designed to provide ease of use, making Dev and Ops lives easier. Let's deploy our application on OpenShift so you can understand some of its functionalities.

Go to openshift.com/ and create a free trial account. This account allows you to experiment OpenShift online with a limit of 1GB of RAM and 1GB of persistent storage. Once you have activated your free OpenShift Online Starter subscription, your account will be provisioned in seconds. Refresh the page and open the web console for OpenShift.

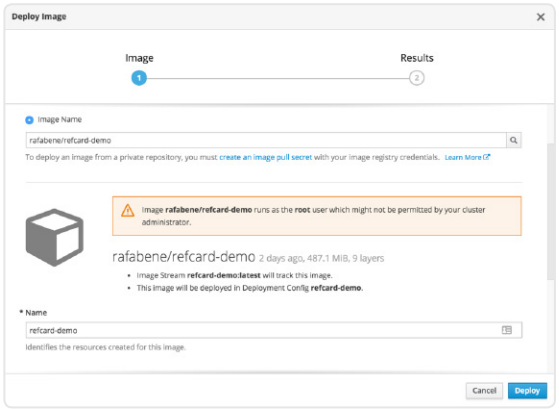
On the top right-hand side of the console, click **Create Project**. The name is the only required field. You can call it *project*.



After the creation of your project, you should see its name. Click on it, and you will land on the page to get started with your project.



To deploy our previously created image, click **Deploy image**. This will open a wizard that allows you to provide the name of the image pushed to Docker Hub. After you type the name of the image including in the format */refcard-demo*, click the magnifier icon so OpenShift can read the metadata of the image. Now, click the blue **Deploy** button.



The image will be downloaded, and a new container instance will be executed based on your image.

The up and down arrows control the number of replicas. Because we are running with a free subscription, scaling to more than two replicas will exceed the usage memory quota of 1GB and no replicas will be created. However, this feature can give you an idea of how easy is to create multiple container replicas and scale your application.

MAKING YOUR CLOUD APPLICATION ACCESSIBLE EXTERNALLY

At this moment, your application is being executed inside the cluster, but it's not accessible externally.

Because we can have more than one container executing, we need to create a load balancer to receive requests to this

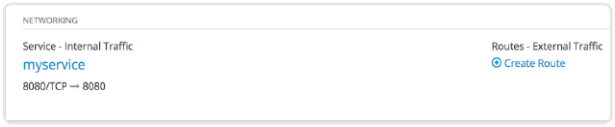
application. This load balancer is known as a "service" in the Kubernetes world. Kubernetes services can be created through a command line tool or using YAML or JSON format.

To avoid the installation of any command line tool, you can import the following YAML file by clicking **Add to project** on the top right-hand side of the screen and selecting **Import YAML/JSON**.

This file can also be copied from here.

```
kind: Service
apiVersion: v1
metadata:
 name: myservice
spec:
 type: LoadBalancer
 ports:
   - port: 8080
 selector:
  app: refcard-demo
```

This file requests the OpenShift/Kubernetes cluster creates a load balancer for the application called *refcard-demo* on port 8080. After defining this Kubernetes service, you will notice that a new section called **Networking** is available in your application. Although the Kubernetes service acts as a load balancer for internal traffic, we need an extra step to make our application available externally. To allow external access, we need to create a route. Click **Create Route**.



In the **Create Route** screen, accept the default values and click the blue **Create** button. Now, on the **Overview** screen, there's a route to your application. You can click on the route link to open the application address in another tab. Don't forget to add */api/hello* in the address to allow it to access the endpoint that we have previously created (*http:///api/hello*).

IMPROVING THE DEVELOPER WORKFLOW

As developer, to place your application in the cloud, you had to follow these steps:

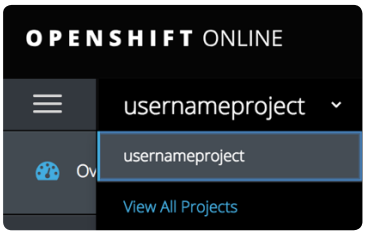
- Create your application.
- Create a Dockerfile.
- Create a Docker image.
- Deploy the image on OpenShift.
- Create a service to act as a load balancer.
- Create a router.

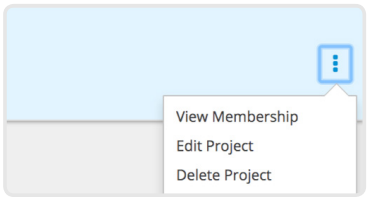
What if I told you that OpenShift has a feature called S2I (source-to-image) that allows your application to be containerized and deploy from the sources? Your application's source code just needs to be accessible in a Git repository on the internet.

If you don't want to create a repository from the scratch for this application, just fork my repository.

Before exploring the S2I features, we need first to get rid of all objects that have been created up until now (deployment, route, service, etc.). The easiest way to do this is by deleting the entire project.

To delete the entire project, select **View all projects** in the top left corner. In the **My projects** screen, select your application menu and click **Delete Project**. You will be asked to confirm the deletion of the project.





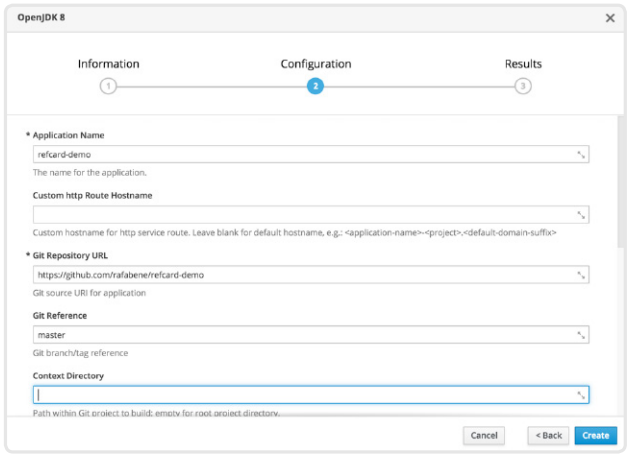
Create your project again. If you receive a message saying that your project already exists, wait a few seconds and try again. The project may still be being removed from the cluster.

In **Get started with your project**, instead of deploying an image as we did previously, click **Browse Catalog**. The OpenShift catalog has support for several languages, databases, and middleware runtimes. You can just go to the **Java** tab and select **OpenJDK 8**.

In the wizard, make sure to fill the following fields:

- **Application name**: refcard-demo
- **Custom HTTP route hostname**:
- **Git repository URL**: Use your forked repository from here.
- **Git reference (git branch)**: Master
- **Context directory**: (Delete the existing value)

Click **Create**.

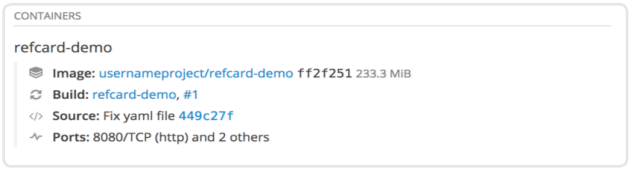


The build process will start automatically. You can check the build log running in the **Overview** page and note that OpenShift will clone your Git repository and run a Maven package on your project. When the build process completes, the new container will be deployed automatically.

You can notice also that the route and service objects were automatically created.

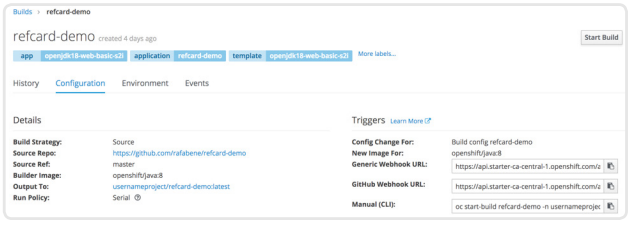
CONTINUOUSLY DELIVERING YOUR JAVA APPLICATION

Since you created your application from the source code, OpenShift also shows the commit version that was used to create your application.



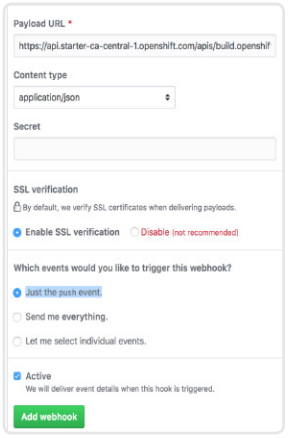
You can enable the continuous delivery of your code by configuring GitHub to send notifications of new commits to OpenShift. When OpenShift receives that notification, it will start the build automatically.

Click the **Build** link, as you can see in the image above. That will take you to **Build Configuration**. In the **Configuration** tab, you can copy the GitHub webhook URL (located on the right-hand side of the screen under the **Triggers** section).



Copy the GitHub webhook URL. Open your GitHub project settings. Click **Add webhook** and fill the web form the following values:

- **Payload URL**:
- **Content type**:
- **Secret**:
- **SSL verification**:
- **Which events would you like to trigger this webhook?**: Just the push event.
- **Active**:



Click the green **Add webhook** button.

To test the continuous delivery, you can modify the file *HelloController.java*. Change, for example, the `RESPONSE_STRING_FORMAT` variable from *Hello from '%s': %d to* Version 2 - Hello from '%s': %d.

When you push the committed change to your GitHub repository, GitHub will notify OpenShift and Build #2 will start automatically.

At the end of the build, OpenShift will replace all the running replicas with the one containing your changes. At the end of the deployment, you can open your browser at *http:///api/hello* and you should see the prefix *Version 2*.

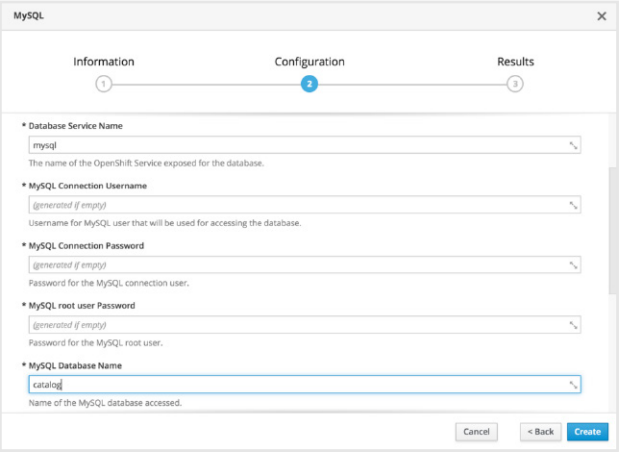
CONNECTING YOUR CONTAINERIZED APPLICATION TO A CONTAINERIZED DATABASE

The application that you forked from https://github.com/rafabene/refcard-demo contains an endpoint that replies with a catalog of items, their names, and their prices in JSON format. These values are stored in an H2 "in-memory" database.

You can check this endpoint by opening the *http:///api/catalog/* endpoint in the browser.

However, an in-memory database is not suitable for Enterprise applications. To replace our H2 database, we need to deploy a MySQL database. Because of the quota limit, before creating the database, make sure that you have only one replica of your *refcard-demo* application running.

In **Search Catalog**, select **MySQL**. On the wizard screen, change the value of *MySQL Database Name* to *catalog*. The other values like "username" and "password" can be generated automatically so you can see how your application can consume those generated values later.



Click **Create**.

MySQL will be deployed, and all the sensitive information is stored in a "secret" object. This object can be consumed by the *refcard-demo* application as an environment variable or as a volume.

To configure our *refcard-demo* application to point to this MySQL instance, click **Deployment Config** for *refcard-demo*. Under the **Environment** tab, add the following values:

- **SPRING_DATASOURCE_URL**: jdbc:mysql://mysql/catalog?useSSL=false
- **SPRING_DATASOURCE_DRIVER_CLASS_NAME**: com.mysql.jdbc.Driver

Now, click **Add Value From Config Map or Secret** and add:

- **SPRING_DATASOURCE_USERNAME**: mysql (secret) / database-user
- **SPRING_DATASOURCE_PASSWORD**: mysql (secret) / database-password

Click **Save**. Because these environment variables were introduced, OpenShift will replace the existing replica with a new one containing these values. After the deployment, you can check your application at *http:///api/catalog/* to verify that it's working.

According to Spring Boot configuration, these environment variables have a higher preference to configure the application over the existing *application.properties* file that already exists in the application.

We don't know the value of the generated username and password for MySQL. However, we could configure the application to read the sensitive values from the "secret" object called *mysql* that was created when we deployed the database.

ACCESSING THE IN-BROWSER TERMINAL FOR THE RUNNING CONTAINERS

You can open a command prompt on any running container on OpenShift. Let's open a terminal on MySQL so we can check if the database has been created and the values has been imported.

To open a terminal, click the pod number for your MySQL application that you see in the **Overview** page. This will open a page containing the details of your pod, which is how Kubernetes/OpenShift refers to the group of containers that run together. Once you click the **Terminal** tab, you will see a web terminal open in the running container *mysql*.

The credentials (username and password) in the MySQL container are stored in environment variables. You can connect to MySQL Server executing `mysql -uecho $MYSQL_USER--password=echo $MYSQL_PASSWORD-h 127.0.0.1 catalog`, as you can see below:

```
sh-4.2$ mysql -u `echo $MYSQL_USER` --password=`echo
$MYSQL_PASSWORD` -h 127.0.0.1 catalog
mysql: [Warning] Using a password on the command line
interface can be insecure.
Welcome to the MySQL monitor.  Commands end with ; or
\g.
Your MySQL connection id is 1307
Server version: 5.7.21 MySQL Community Server (GPL)

Copyright (c) 2000, 2018, Oracle and/or its affiliates.
All rights reserved.

Oracle is a registered trademark of Oracle Corporation
and/or its
affiliates. Other names may be trademarks of their
respective owners.
```

CODE BLOCK CONTINUED ON FOLLOWING PAGE

```
Type 'help;' or '\h' for help. Type '\c' to clear the
current input statement.

mysql>
```

Once you are connected, you can check the values in the database.

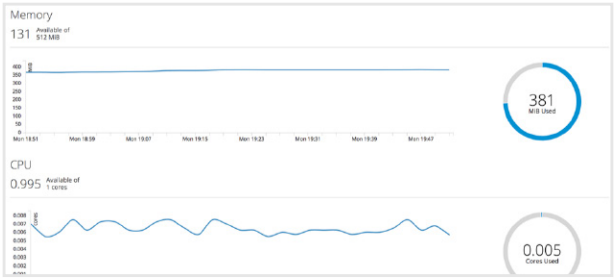
```
mysql> show tables;
+--------------------+
| Tables_in_catalog |
+--------------------+
| PRODUCT            |
+--------------------+
1 row in set (0.00 sec)

mysql> select * from PRODUCT;
. . .
8 rows in set (0.00 sec)

mysql> exit
Bye
sh-4.2$
```

MONITORING THE CPU, MEMORY, AND I/O UTILIZATION OF THE RUNNING CONTAINERS

On each pod, the user can monitor the CPU, memory, and I/O of the running containers. This can be done by clicking on the **Metrics** tab. A pre-defined date range can be selected to view CPU, memory, and I/O historically.



Written by **Rafael Benevides**, Director of Developer Experience at Red Hat

Rafael Benevides is Director of Developer Experience at Red Hat. With many years of experience in several fields of the IT industry, he helps developers and companies all over the world to be more effective in software development. Rafael considers himself a problem-solver who has a big love for sharing. He is a member of Apache DeltaSpike PMC, a Duke's Choice Award Winner project, and a speaker at conferences like JavaOne, Devoxx, TDC, DevNexus, and many others. You can learn more about him on his website: rafabene.com. Follow him on his Linkedin an Twitter accounts.

DZone
A DEVADA MEDIA PROPERTY

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheat sheets, research guides, feature articles, source code and more. "DZone is a developer's dream," says PC Magazine.

DZone, Inc.
150 Preston Executive Dr. Cary, NC 27513
888.678.0399 919.678.0300