Caching SpringBoot Microservices with Hazelcast in Kubernetes

You can see the whole project here.

What you’ll learn

You will learn how to use Hazelcast distributed caching with SpringBoot and deploy to a local Kubernetes cluster. You will then create a Kubernetes Service which load balance between containers and verify that you can share data between Microservices.

The microservice you will deploy is called `hazelcast-spring`. The `hazelcast-spring` microservice simply helps you put a data and read it back. As Kubernetes Service will send the request to different pod each time you initiate the request, the data will be served by shared hazelcast cluster between `hazelcast-spring` pods.

You will use a local single-node Kubernetes cluster. However, you can deploy this application on any kubernetes distributions.

What is Hazelcast?

Hazelcast is an open source In-Memory Data Grid (IMDG). It provides elastically scalable distributed In-Memory computing, widely recognized as the fastest and most scalable approach to application performance.

Hazelcast is designed to scale up to hundreds and thousands of members. Simply add new members and they will automatically discover the cluster and will linearly increase both memory and processing capacity.

Why Spring Boot?

Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can "just run". To learn more about Spring Boot, visit website.

Prerequisites

Before you begin, have the following tools installed:

First, you will need Apache Maven to build and run the project.

Also you will need a containerization software for building containers. Kubernetes supports a variety of container types. You will use `Docker` in this guide. For installation instructions, refer to the official Docker documentation.

Windows | Mac

Use Docker Desktop, where a local Kubernetes environment is pre-installed and enabled. If you do not see the Kubernetes tab then you have an older version of Docker Desktop; upgrade to the latest version.

Complete the setup for your operating system:

- Set up Docker for Windows. On the Docker for Windows *General Setting* page, ensure that the option `Expose daemon on tcp://localhost:2375 without TLS` is enabled. This is required by the `dockerfile-maven` part of the build.
- Set up Docker for Mac.
- After following one of the sets of instructions, ensure that Kubernetes (not Swarm) is selected as the orchestrator in Docker Preferences.

Linux

You will use `Minikube` as a single-node Kubernetes cluster that runs locally in a virtual machine. For Minikube installation instructions see the minikube installation instructions. Make sure to pay attention to the requirements as they vary by platform.

Getting started

The fastest way to work through this guide is to clone the Git repository and use the project provided inside:

```
$ > git clone https://github.com/hazelcast-guides/caching-springboot-microservices-on-
kubernetes
$ > cd caching-springboot-microservices-on-kubernetes
```

The `initial` directory contains the starting project that you will build upon.

The `final` directory contains the finished project that you will build.

Running Spring Application

The application in initial directory is a basic SpringBoot app having 3 endpoints:

- **"/"** is the homepage returning "Welcome" string only
- **"/put"** is the page where key and values can be put on a concurrent hash map.
- **"/get"** is the page where the values in the map can be obtained by keys.

Build the app using Maven in the `initial` directory:

```
$ > mvn package
```

Run the application:

```
$ > java -jar target/hazelcast-spring-app-0.1.0.jar
```

Now your app is runnning on localhost:8080. You can test by following requests:

```
$ > curl "localhost:8080"
$ > curl "localhost:8080/put?key=key1&value=hazelcast"
$ > curl "localhost:8080/get?key=key1"
```

This part was the introduction of the application. You can stop your application by CTRL + C.

Dockerizing the App

To create the docker image of the application, add following line into pom.xml file.

```
<!-- add among other properties -->
<properties>
    <docker.image.prefix>springio</docker.image.prefix>
</properties>

<!-- add among other plugins -->
<plugins>
<plugin>
      <groupId>com.spotify</groupId>
      <artifactId>dockerfile-maven-plugin</artifactId>
      <version>${version.dockerfile-maven-plugin}</version>
      <configuration>
         <repository>${docker.image.prefix}/${project.artifactId}</repository>
      </configuration>
</plugin>
</plugins>
```

Then create the Dockerfile under `initial` directory named "Dockerfile" containing the instructions for creating a docker image:

```
FROM openjdk:8-jdk-alpine

VOLUME /tmp

ARG JAR_FILE=target/hazelcast-spring-app-0.1.0.jar

ADD ${JAR_FILE} hazelcast-spring-demo.jar

ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/hazelcast-
spring-demo.jar"]
```

Before creating the Docker image of the app, first rebuild the app:

```
$ > mvn clean package
```

Then create image file:

```
$ > docker build -t hazelcast-spring-demo .
```

Now, the image must be seen among the docker images:

```
$ > docker images

REPOSITORY                           TAG                 IMAGE ID            CREATED
SIZE
hazelcast-spring-demo                latest              5f65a62b0aa0        19
seconds ago      122MB
openjdk                              8-jdk-alpine        a3562aa0b991        5 weeks
ago         105MB
k8s.gcr.io/kube-proxy-amd64          v1.10.11            7387003276ac        6 months
ago        98.3MB
k8s.gcr.io/kube-apiserver-amd64      v1.10.11            e851a7aeb6e8        6 months
ago        228MB
```

Running the app in container

Now that the Docker image is ready, check if the image runs properly:

```
$ > docker run -p 5000:8080 hazelcast-spring-demo
```

Test the app on the port 5000:

```
$ > curl "localhost:5000"
$ > curl "localhost:5000/put?key=key1&value=hazelcast"
$ > curl "localhost:5000/get?key=key1"
```

If you see the same responses as the ones you get when the app is run without container, that means it’s all OK with the image. To stop the container, get the container ID first:

```
$ > docker ps
```

Then find the application’s container ID and stop the container:

```
$ > docker stop [CONTAINER-ID]
```

Starting and preparing your cluster for deployment

Now that you have a proper docker image, deploy the app to kuberntes pods. Start your Kubernetes cluster first.

Windows | **Mac**

Start your Docker Desktop environment. Make sure "Docker Desktop is running" and "Kubernetes is running" status are updated.

Linux

Run the following command from a command line:

```
$ > minikube start
```

Validate Kubernetes environment

Next, validate that you have a healthy Kubernetes environment by running the following command from the command line.

```
$ > kubectl get nodes
```

This command should return a `Ready` status for the master node.

Windows | **Mac**

You do not need to do any other step.

Linux

Run the following command to configure the Docker CLI to use Minikube's Docker daemon. After you run this command, you will be able to interact with Minikube's Docker daemon and build new images directly to it from your host machine:

```
$ > eval $(minikube docker-env)
```

After you're sure that a master node is ready, create kubernetes.yaml under initial directory with the same content in the final/kubernetes.yaml file. This file defines two Kubernetes resources: one statefulset and one service. StatefulSet is preferred solution for Hazelcast because it enables controlled scale out/in of your microservices for easy data distribution. To learn more about StatefulSet, you can visit Kubernetes documentation.

By default, we create 2 replicas of `hazelcast-spring` microservice behind the `hazelcast-spring-service` which forwards requests to one of the pods available in the kubernetes cluster.

`MY_POD_NAME` is an environment variable made available to the pods so that each microservice knows which pod they are in. This is going to be used in this guide in order to show which pod is responding to the http request.

Run the following command to deploy the resources as defined in kubernetes.yaml:

```
$ > kubectl apply -f kubernetes.yaml
```

Run the following command to check the status of your pods:

```
$ > kubectl get pods
```

You'll see an output similar to the following if all the pods are healthy and running:

```
NAME                             READY     STATUS     RESTARTS    AGE
hazelcast-spring-statefulset-0   1/1       Running    0           7s
hazelcast-spring-statefulset-1   1/1       Running    0           5s
```

Send request to port :31000 and see the pods responding.

```
$ > curl localhost:31000
```

And add a value to the map and then get the value:

```
$ > curl "localhost:31000/put?key=key1&value=hazelcast"

{"value":"hazelcast","podName":"hazelcast-spring-statefulset-1"} --> the second pod's
response

$ > while true; do curl localhost:31000/get?key=key1;echo; sleep 2; done

{"value":"hazelcast","podName":"hazelcast-spring-statefulset-1"} --> the second pod's
response
{"value":null,"podName":"hazelcast-spring-statefulset-0"} --> the first pod's response
{"value":"hazelcast","podName":"hazelcast-spring-statefulset-1"} --> the second pod's
response
{"value":null,"podName":"hazelcast-spring-statefulset-0"} --> the first pod's response
```

As can be seen, data is not shared between nodes. Here is where Hazelcast comes into action. Kill active pods under `initial` directory by:

```
$ > kubectl delete -f kubernetes.yaml
```

Hazelcast Caching among Kubernetes pods

Now we will use Hazelcast Caching among the pods. Update the pom.xml file by adding those dependencies:

```
<dependency>
            <groupId>com.hazelcast</groupId>
            <artifactId>hazelcast</artifactId>
            <version>${version.hazelcast}</version>
</dependency>
<dependency>
            <groupId>com.hazelcast</groupId>
            <artifactId>hazelcast-kubernetes</artifactId>
            <version>${version.hazelcast-kubernetes}</version>
</dependency>
```

Then modify the CommandController.java such that Hazelcast is used in the map. Also add Hazelcast config to Application.java file and hazelcast libraries as well. Those versions are the ones under `final` folder.

Rebuild the app and create new image:

```
$ > mvn clean package
$ > docker build -t hazelcast-spring-demo .
```

Before deploying on kubernetes, create rbac.yaml file as in the `final` directory. Role Based Access Controller(RBAC) configuration is used to give access to Kubernetes Master API from pods which runs microservices. Hazelcast requires a read access to autodiscover other hazelcast members and form hazelcast cluster.

Run the following commands to deploy the resources as defined in kubernetes.yaml and rbac.yaml in the specified order:

```
$ > kubectl apply -f rbac.yaml
$ > kubectl apply -f kubernetes.yaml
```

Run the following command to check the status of your pods:

```
$ > kubectl get pods
```

You should also check if hazelcast cluster is formed by checking one of the pod's log file:

```
$ > kubectl logs hazelcast-spring-statefulset-1
```

You must see such a response at the end of the log:

```
Members {size:2, ver:2} [
    Member [10.1.0.52]:5701 - ac54036d-c16f-40ae-9531-93e6f0683cf9 this
    Member [10.1.0.53]:5701 - d963bb82-3842-49fd-a522-82c8543bdb9d
]
```

If it's not seen, wait for pods to be configured and try again.

Now we expect all nodes to give the same value for the same key put on the map via one pod only. Let's try:

```
$ > curl "http://localhost:31000/put?key=key1&value=hazelcast"

    {"value":"hazelcast","podName":"hazelcast-spring-statefulset-1"} --> pod1
responsed

$ > while true; do curl localhost:31000/get?key=key1;echo; sleep 2; done

    {"value":"hazelcast","podName":"hazelcast-spring-statefulset-1"} --> pod1
responsed
    {"value":"hazelcast","podName":"hazelcast-spring-statefulset-1"} --> pod1
responsed
    {"value":"hazelcast","podName":"hazelcast-spring-statefulset-0"} --> pod0
responsed
```

As can be seen, the insertion is made on hazelcast-spring-statefulset-1 but both nodes gives the same value for the key now.

Scaling with Hazelcast

Scale the cluster with one more pod and see that you still retrieve the shared data.

```
$ > kubectl scale statefulset hazelcast-spring-statefulset --replicas=3
```

Run following command to see the latest status of the pods

```
$ > kubectl get pods
```

As you can see, a new pod `hazelcast-spring-statefulset-2` has joined to the cluster.

```
NAME                              READY     STATUS    RESTARTS   AGE
hazelcast-spring-statefulset-0   1/1       Running   0          8m
hazelcast-spring-statefulset-1   1/1       Running   0          8m
hazelcast-spring-statefulset-2   1/1       Running   0          31s
```

Run the following command again to see the output

```
$ > while true; do curl http://localhost:31000/get?key=key1;echo; sleep 2; done
```

```
{"value":"hazelcast","podName":"hazelcast-spring-statefulset-2"}
{"value":"hazelcast","podName":"hazelcast-spring-statefulset-0"}
{"value":"hazelcast","podName":"hazelcast-spring-statefulset-2"}
{"value":"hazelcast","podName":"hazelcast-spring-statefulset-1"}
```

As you can see, `hazelcast-caching-statefulset-2` is returning correct data.

Testing microservices running on Kubernetes

Create a testing class under `initial/src/test/java/it/io/spring/guides/hazelcast/` named `HazelcastCachingIT.java` .The contents of the test file is available under `final` directory.

The test makes sure that the **/put** endpoint is handled by one pod and **/get** methods returns the same data from the other kubernetes pod.

It first puts a key/value pair to hazelcast-spring microservice and keeps podname in the firstpod variable. In the second part, tests submits multiple **/get** requests until to see that podname is different then the pod which initially handled /**put** request.

In order to run integration tests, you must have a running hazelcast-spring microservices in minikube environment. As you have gone through all previous steps, you already have it.

Navigate back to `initial` directory and run the test:

```
$ > mvn -Dtest=HazelcastCachingIT test
```

If the tests pass, you'll see a similar output to the following:

```
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running HazelcastCachingIT
10:12:27.087 [Time-limited test] DEBUG org.springframework.web.client.RestTemplate -
HTTP GET http://localhost:31000/put?key=key1&value=hazelcast-spring-guide
10:12:27.175 [Time-limited test] DEBUG org.springframework.web.client.RestTemplate -
Accept=[application/json, application/*+json]
10:12:27.312 [Time-limited test] DEBUG org.springframework.web.client.RestTemplate -
Response 200 OK
...
...
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 5.354 s - in
HazelcastCachingIT
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
```

Tearing down the environment

When you no longer need your deployed microservices, you can delete all Kubernetes resources by running the kubectl delete command: You might need to wait up to 30 seconds as stateful sets kills pods one at a time.

```
$ > kubectl delete -f kubernetes.yaml
```

Windows | Mac

Nothing more needs to be done for Docker Desktop.

Linux

Perform the following steps to return your environment to a clean state.

1. Point the Docker daemon back to your local machine:

   ```
   $ > eval $(minikube docker-env -u)
   ```

2. Stop your Minikube cluster:

   ```
   $ > minikube stop
   ```

3. Delete your cluster:

   ```
   $ > minikube delete
   ```

Great work! You're done!

You have just run a SpringBoot application and created its Docker image. First you runned the app on a container and then deployed it to Kubernetes. You then added Hazelcast caching to the hazelcast-spring, tested with a simple curl command. You also scaled out the microservices and saw that data is shared between microservices. As a last step, you ran integration tests against hazelcast-spring that was deployed in a Kubernetes cluster.