Caching Micronaut microservices on Kubernetes using Hazelcast

This guide will get you started in using Hazelcast in Micronaut applications on Kubernetes.

You can see the whole project here.

What you'll learn

You will learn how to use Hazelcast distributed caching with Micronaut 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-micronaut`. The `hazelcast-micronaut` 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-micronaut` pods.

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

Why Micronaut

Micronaut is a modern, JVM-based, full stack microservices framework designed for building modular, easily testable microservice applications. It comes with different options to share data among the running services, such as caching.

This guide contains a basic Micronaut microservice code sample using Hazelcast IMDG. You can see the whole project here and start building your app in the final directory. However, this guide will start from an initial point and build the application step by step.

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-micronaut-microservices-on-kubernetes
$ > cd caching-micronaut-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 the Micronaut Application

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

- `/` is the homepage returning “Homepage” string only
- `/put` is the mapping where key and value is saved to a local map through `@CachePut` annotation.
- `/get` is the mapping where the values in the local map can be obtained by keys through `@Cacheable` annotation.

You can run this application using the commands below:

```
$ mvn clean package
$ java -jar target/caching-micronaut-microservices-on-kubernetes-0.1.0.jar
```

Now your app is running at `localhost:8080`. You can test it by using the following on another console:

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

The output should be similar to the following:

```
{"value":"hazelcast"}
```

The value returns as `hazelcast` since we put it in the second command, and `podName` returns `null` because we are not running the application in k8s environment yet. After the testing, you can kill the running application on your console.

Running the App in a Container

To create the container (Docker) image of the application, we will use Jib tool. It allows to build containers from Java applications without a Docker file or even changing the `pom.xml` file. To build the image, you can run the command below:

```
$ mvn clean compile com.google.cloud.tools:jib-maven-plugin:1.8.0:dockerBuild
```

This command will compile the application, create a Docker image, and register it to your local container registry. Now, we can run the container using the command below:

```
$ docker run -p 5000:8080 caching-micronaut-microservices-on-kubernetes:0.1.0
```

This command runs the application and binds the local `5000` port to the `8080` port of the container. Now, we should be able to access the application using the following commands:

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

The results will be the same as before. You can kill the running application after testing. Now, we have a container image to deploy on k8s environment.

Running the App in Kubernetes Environment

To run the app on k8s, we need a running environment. As stated before, we will be using `minikube` for demonstration.

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)
```

We will use a deployment configuration which builds a service with two pods. Each of these pods will run one container which is built with our application image. You can see our example configuration file (named as `kubernetes.yaml`) in the repository. Please click here to download this file. You can see that we are also setting an environment variable named `MY_POD_NAME` to reach the pod name in the application.

After downloading the configuration file and putting it under initial directory, you can deploy the containers using the command below:

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

Now, we should have a running deployment. You can check if everything is alright by getting the pod list:

```
$ kubectl get pods
```

It is time to test our running application. But first, we need the the IP address of the running k8s cluster to access it. Since we are using `minikube`, we can get the cluster IP using the command below:

```
$ minikube ip
```

We have the cluster IP address, thus we can test our application:

```
$ curl "http://[CLUSTER-IP]:31000/put?key=myKey&value=hazelcast"
$ while true; do curl [CLUSTER-IP]:31000/get?key=myKey;echo; sleep 2; done
```

The second command makes a request in a loop in order to see the responses from both pods. At some point, you should see a result similar below:

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

This means the pod named `hazelcast-micronaut-statefulset-1` got the put request, and stored the value in its local cache. However, the other pod couldn't get this data and displayed `null` since there are no distributed caching between pods. To enable distributed caching, we will use Hazelcast IMDG in the next step. Now, you can delete the deployment using the following command:

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

Caching using Hazelcast

To configure caching with Hazelcast, firstly we will add some dependencies to our `pom.xml` file:

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

The first dependency is for Micronaut Cache for Hazelcast, the second one is for Hazelcast IMDG itself, and the last one is for Hazelcast's Kubernetes Discovery Plugin which helps Hazelcast members to discover each other on a k8s environment. Now, we just need to add a configuration bean to enable Hazelcast:

```
@Singleton
public class HazelcastAdditionalSettings
        implements BeanCreatedEventListener<HazelcastMemberConfiguration> {

    public HazelcastMemberConfiguration
onCreated(BeanCreatedEvent<HazelcastMemberConfiguration> event) {
        HazelcastMemberConfiguration configuration = event.getBean();
        configuration.getGroupConfig().setName("micronaut-cluster");
        JoinConfig joinConfig = configuration.getNetworkConfig().getJoin();
        joinConfig.getMulticastConfig().setEnabled(false);
        joinConfig.getKubernetesConfig().setEnabled(true);
        return configuration;
    }
}
```

This bean creates a `HazelcastAdditionalSettings` object to configure Hazelcast members. We enable the k8s config for discovery.

Our application with Hazelcast caching is now ready to go. We do not need to change anything else because we are already using Micronaut caching annotations in `CommandService` class.

Running the App with Hazelcast IMDG in Kubernetes Environment

Before deploying our updated application on k8s, you should create a `rbac.yaml` file which you can find it here. This is the role-based access control (RBAC) configuration which is used to give access to the Kubernetes Master API from pods. Hazelcast requires read access to auto-discover other Hazelcast members and form Hazelcast cluster. After creating/downloading the file, apply it using the command below:

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

Now, we can build our container image again and deploy it on k8s:

```
$ mvn clean compile com.google.cloud.tools:jib-maven-plugin:1.8.0:dockerBuild
$ kubectl apply -f kubernetes.yaml
```

Our application is running, and it is time to test it again:

```
$ curl "http://[CLUSTER-IP]:31000/put?key=myKey&value=hazelcast"
$ while true; do curl [CLUSTER-IP]:31000/get?key=myKey;echo; sleep 2; done
```

Since we configured distributed caching with Hazelcast, we can see the output similar to the following:

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

Now, both of the pods have the same value and our data is retrieved from Hazelcast distributed cache through all microservices!

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
```

Conclusion

In this guide, we first developed a simple microservices application which uses caching with Micronaut annotations. The usage is very simple, but the cached data is not accessible from all microservices on a k8s environment if we don't use a distributed cache. In order to enable distributed caching, we used Hazelcast IMDG. The configuration is easy, just by adding a configuration bean. In the end, we succeeded to enable distributed caching among pods which

helped us to access the same cached data from all microservices.