AWS Architecture Blog

Field Notes: Running a Stateful Java Service on Amazon EKS

by Tom Cheung and Bastian Klein | on 22 FEB 2021 | in Amazon Elastic Kubernetes Service, Architecture, Containers, Technical How-to | Permalink | Share

This post was co-authored by Tom Cheung, Cloud Infrastructure Architect, AWS Professional Services and Bastian Klein, Solutions Architect at AWS.

Containerization helps to create secure and reproducible runtime environments for applications. Container orchestrators help to run containerized applications by providing extended deployment and scaling capabilities, among others. Because of this, many organizations are installing such systems as a platform to run their applications on. Organizations often start their container adaption with new workloads that are well suited for the way how orchestrators manage containers.

After they gained their first experiences with containers, organizations start migrating their existing applications to the same container platform to simplify the infrastructure landscape and unify their deployment mechanisms. Migrations come with some challenges, as the applications were not designed to run in a container environment. Many of the existing applications work in a stateful manner. They are persisting files to the local storage and make use of stateful sessions. Both requirements need to be met for the application to properly work in the container environment.

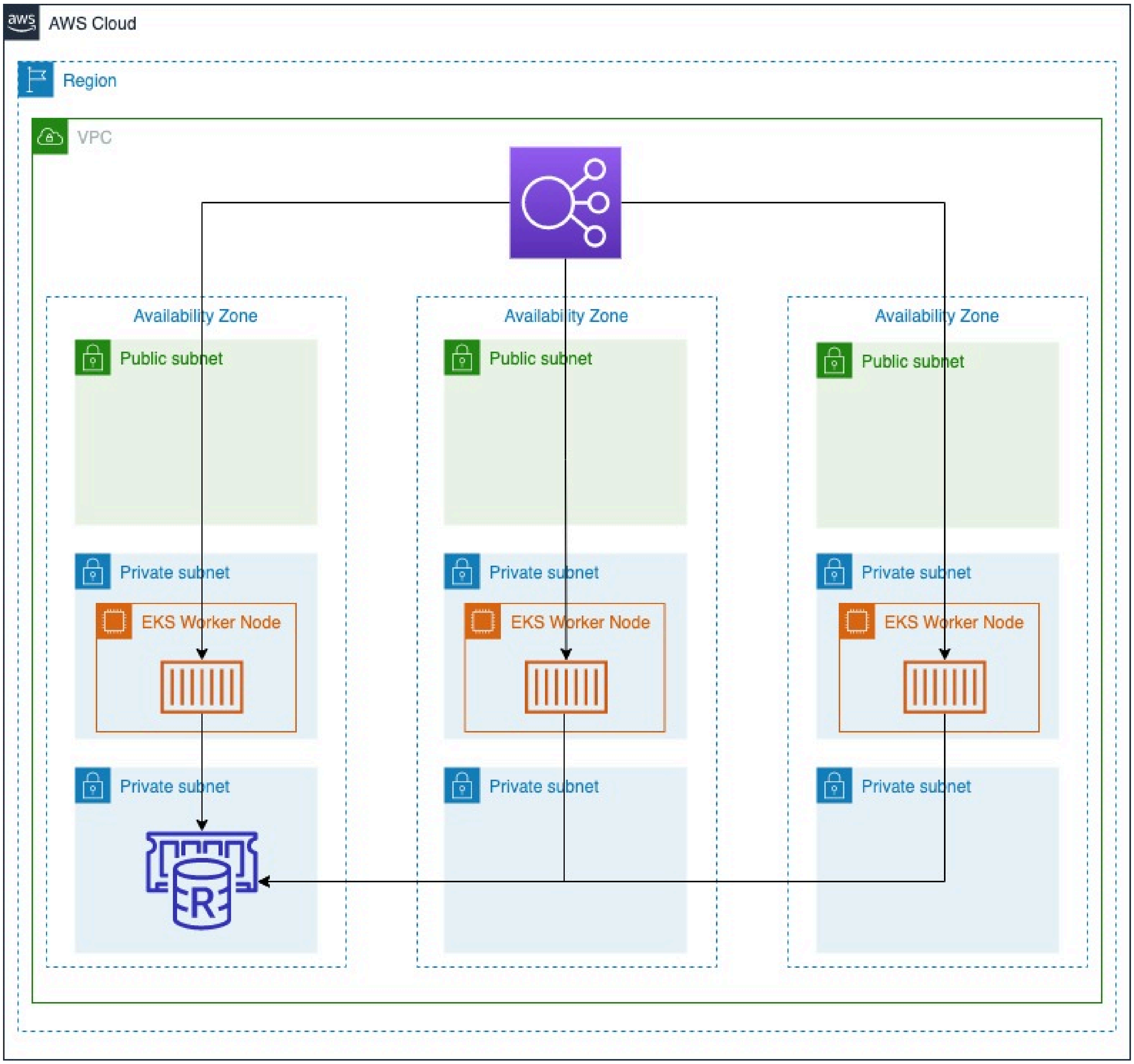
This blog post shows how to run a stateful Java service on Amazon EKS with the focus on how to handle stateful sessions You will learn how to deploy the service to Amazon EKS and how to save the session state in an Amazon ElastiCache Redis database. There is a GitHub Repository that provides all sources that are mentioned in this article. It contains AWS CloudFormation templates that will setup the required infrastructure, as well as the Java application code along with the Kubernetes resource templates.

The Java code used in this blog post and the GitHub Repository are based on a Blog Post from Java In Use: Spring Boot + Session Management Example Using Redis. Our thanks for this content contributed under the MIT-0 license to the Java In Use author.

Overview of architecture

Kubernetes is a popular Open Source container orchestrator that is widely used. Amazon EKS is the managed Kubernetes offering by AWS and used in this example to run the Java application. Amazon EKS manages the Control Plane for you and gives you the freedom to choose between self-managed nodes, managed nodes or AWS Fargate to run your compute.

The following architecture diagram shows the setup that is used for this article.



- There is a VPC composed of three public subnets, three subnets used for the application and three subnets reserved for the database.
- For this application, there is an Amazon ElastiCache Redis database that stores the user sessions and state.
- The Amazon EKS Cluster is created with a Managed Node Group containing three t3.micro instances per default. Those instances run the three Java containers.
- To be able to access the website that is running inside the containers, Elastic Load Balancing is set up inside the public subnets.

- The Elastic Load Balancing (Classic Load Balancer) is not part of the CloudFormation templates, but will automatically be created by Amazon EKS, when the application is deployed.

Walkthrough

Here are the high-level steps in this post:

- Deploy the infrastructure to your AWS Account
- Inspect Java application code
- Inspect Kubernetes resource templates
- Containerization of the Java application
- Deploy containers to the Amazon EKS Cluster
- Testing and verification

Prerequisites

- An AWS account
- The following tools need to be installed:
  - aws cli version 2
  - jq
  - Kubernetes cli
  - Maven
  - Docker
- Basic and advanced knowledge of Kubernetes, Docker, Java, Spring Boot and microservices

If you do not want to set this up on your local machine, you can use AWS Cloud9.

Deploying the infrastructure

To deploy the infrastructure, you first need to clone the Github repository.

```
git clone https://github.com/aws-samples/amazon-eks-example-for-stateful-java-service.git
```

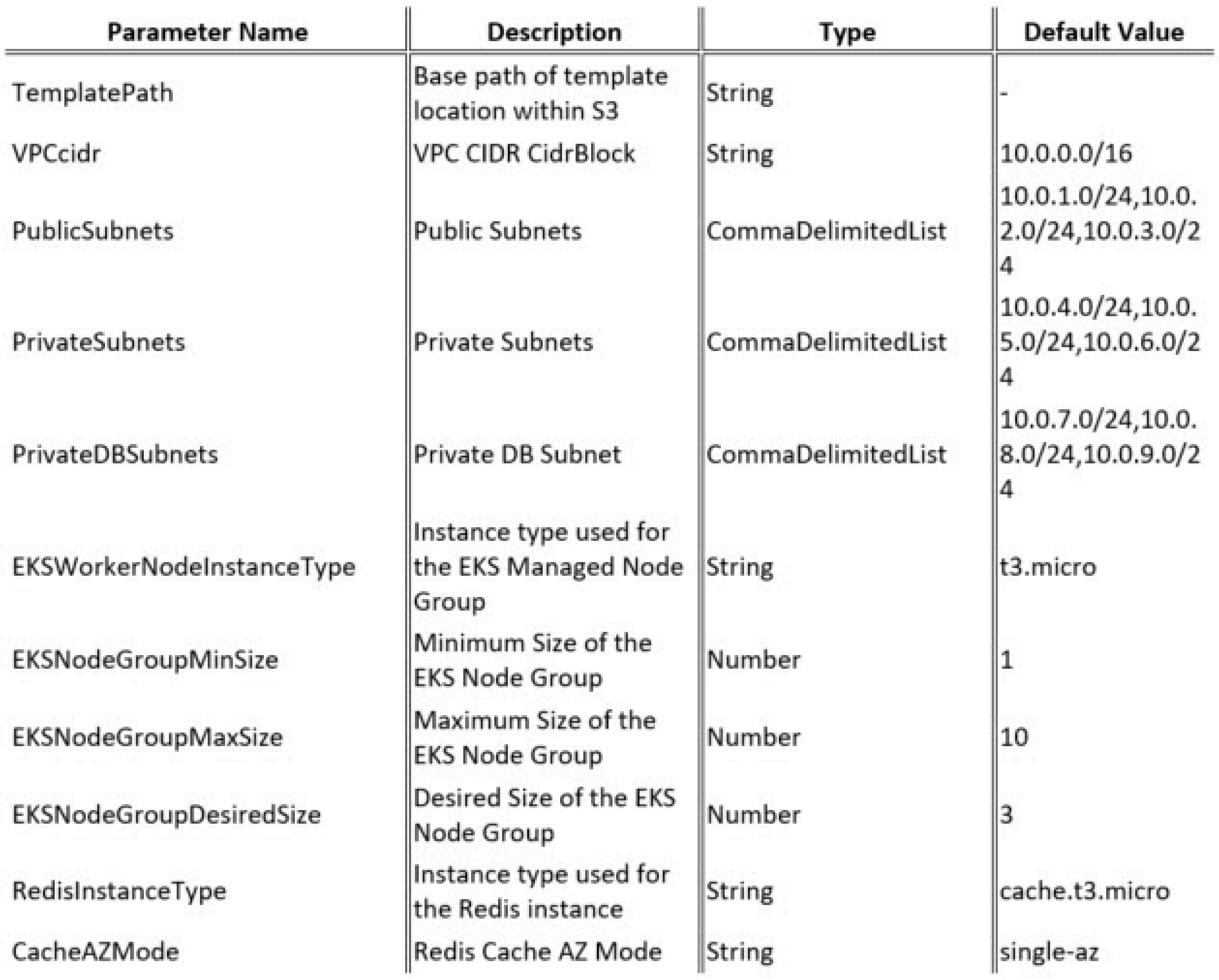
This repository contains a set of CloudFormation Templates that set up the required infrastructure outlined in the architecture diagram. This repository also contains a deployment script deploy.sh that issues all the necessary CLI commands. The script has one required argument -p that reflects the aws cli profile that should be used. Review the Named Profiles documentation to set up a profile before continuing.

If the profile is already present, the deployment can be started using the following command:

```
./deploy.sh -p <profile name>
```

The creation of the infrastructure will roughly take 30 minutes.

The below table shows all configurable parameters of the CloudFormation template:



Parameter Name	Description	Type	Default Value
TemplatePath	Base path of template location within S3	String	-
VPCcidr	VPC CIDR CidrBlock	String	10.0.0.0/16
PublicSubnets	Public Subnets	CommaDelimitedList	10.0.1.0/24,10.0.2.0/24,10.0.3.0/24
PrivateSubnets	Private Subnets	CommaDelimitedList	10.0.4.0/24,10.0.5.0/24,10.0.6.0/24
PrivateDBSubnets	Private DB Subnet	CommaDelimitedList	10.0.7.0/24,10.0.8.0/24,10.0.9.0/24
EKSWorkerNodeInstanceType	Instance type used for the EKS Managed Node Group	String	t3.micro
EKSNodeGroupMinSize	Minimum Size of the EKS Node Group	Number	1
EKSNodeGroupMaxSize	Maximum Size of the EKS Node Group	Number	10
EKSNodeGroupDesiredSize	Desired Size of the EKS Node Group	Number	3
RedisInstanceType	Instance type used for the Redis instance	String	cache.t3.micro
CacheAZMode	Redis Cache AZ Mode	String	single-az

This template is initiating several steps to deploy the infrastructure. First, it validates all CloudFormation templates. If the validation was successful, an Amazon S3 Bucket is created and the CloudFormation Templates are uploaded there. This is necessary because nested stacks are used. Afterwards the deployment of the main stack is initiated. This will automatically trigger the creation of all nested stacks.

Java application code

The following code is a Java web application implemented using Spring Boot. The application will persist session data at Amazon ElastiCache Redis, which enables the app to become stateless. This is a crucial part of the migration, because it allows you to use Kubernetes horizontal scaling features with Kubernetes resources like Deployments, without the need to use sticky load balancer sessions.

This is the Java ElastiCache Redis implementation by Spring Data Redis and Spring Boot. It allows you to configure the host and port of the deployed Redis instance. Because this is environment-specific information, it is not configured in the properties file It is injected as environment variables during runtime.

`/java-microservice-on-eks/src/main/java/com/amazon/aws/Config.java`

JavaScript
```
    blic void setHost(String host) {
        this.host = host;
    }

    public Integer getPort() {
        return port;
    }

    public void setPort(Integer port) {
        this.port = port;
    }

    @Bean
    public LettuceConnectionFactory redisConnectionFactory() {

        return new LettuceConnectionFactory(new RedisStandaloneConfiguration(this.h
    }

}
```

Containerization of Java application

`/java-microservice-on-eks/Dockerfile`

JavaScript
```
FROM openjdk:8-jdk-alpine

MAINTAINER Tom Cheung <email address>, Bastian Klein<email address>
VOLUME /tmp
VOLUME /target

RUN addgroup -S spring && adduser -S spring -G spring
USER spring:spring
ARG DEPENDENCY=target/dependency
COPY ${DEPENDENCY}/BOOT-INF/lib /app/lib
COPY ${DEPENDENCY}/META-INF /app/META-INF
COPY ${DEPENDENCY}/BOOT-INF/classes /app
COPY ${DEPENDENCY}/org /app/org

ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-cp","app:app/lib/*", "c
```

This is the Dockerfile to build the container image for the Java application. OpenJDK 8 is used as the base container image. Because of the way Docker images are built, this sample explicitly does not use a so-called 'fat jar'. Therefore, you have separate image layers for the dependencies and the application code. By leveraging the Docker caching mechanism, optimized build and deploy times can be achieved.

Kubernetes Resources

After reviewing the application specifics, we will now see which Kubernetes Resources are required to run the application.

Kubernetes uses the concept of config maps to store configurations as a resource within the cluster. This allows you to define key value pairs that will be stored within the cluster and which are accessible from other resources.

`/java-microservice-on-eks/k8s-resources/config-map.yaml`

JavaScript
```
apiVersion: v1
kind: ConfigMap
metadata:
  name: java-ms
  namespace: default
data:
  host: "***.***.0001.euc1.cache.amazonaws.com"
  port: "6379"
```

In this case, the config map is used to store the connection information for the created Redis database.

To be able to run the application, Kubernetes Deployments are used in this example. Deployments take care to maintain the state of the application (e.g. number of replicas) with additional deployment capabilities (e.g. rolling deployments).

`/java-microservice-on-eks/k8s-resources/deployment.yaml`

YAML
```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: java-ms
  # labels so that we can bind a Service to this Pod
  labels:
    app: java-ms
spec:
  replicas: 3
  selector:
    matchLabels:
      app: java-ms
  template:
    metadata:
      labels:
        app: java-ms
    spec:
      containers:
```

Deployments are also the place for you to use the configurations stored in config maps and map them to environment variables. The respective configuration can be found under "env". This setup relies on the Spring Bo feature that is able to read environment variables and write them into the according system properties.

Now that the containers are running, you need to be able to access those containers as a whole from within the cluster, but also from the internet. To be able to route traffic cluster internally Kubernetes has a resource called Service. Kubernetes Services get a Cluster internal IP and DNS name assigned that can be used to access all containers that belong to that Service. Traffic will, by default, be distributed evenly across all replicas.

`/java-microservice-on-eks/k8s-resources/service.yaml`

YAML

```
apiVersion: v1
kind: Service
metadata:
  name: java-ms
spec:
  type: LoadBalancer
  ports:
    - protocol: TCP
      port: 80 # Port for LB, AWS ELB allow port 80 only
      targetPort: 8080 # Port for Target Endpoint
  selector:
    app: java-ms
```

The "selector" defines which Pods belong to the services. It has to match the labels assigned to the pods. The labels are assigned in the "metadata" section in the deployment.

Deploy the Java service to Amazon EKS

Before the deployment can start, there are some steps required to initialize your local environment:

1. Update the local kubeconfig to configure the kubectl with the created cluster
2. Update the k8s-resources/config-map.yaml to the created Redis Database Address
3. Build and package the Java Service
4. Build and push the Docker image
5. Update the k8s-resources/deployment.yaml to use the newly created image

These steps can be automatically executed using the init.sh script located in the repository. The script needs following parameter:

1. -u – Docker Hub User Name
2. -r – Repository Name
3. -t – Docker image version tag

A sample invocation looks like this: `./init.sh -u bastianklein -r java-ms -t 1.2`

This information is used to concatenate the full docker repository string. In the preceding example this would resolve to bastianklein/java-ms:1.2, which will automatically be pushed to your Docker Hub repository. If you are not yet logged in to docker on the command line execute `docker login` and follow the displayed steps before executing the `init.sh` script.

As everything is set up, it is time to deploy the Java service. The below list of commands first deploys all Kubernetes resources and then lists pods and services.

```
kubectl apply -f k8s-resources/
```

This will output:

```
configmap/java-ms created
deployment.apps/java-ms created
service/java-ms created
```

Now, list the freshly created pods by issuing `kubectl get pods`.

```
NAME                        READY   STATUS              RESTARTS   AGE

java-ms-69664cc654-7xzkh    0/1     ContainerCreating   0          1s
```

```
java-ms-69664cc654-b9lxb    0/1      ContainerCreating    0          1s
```

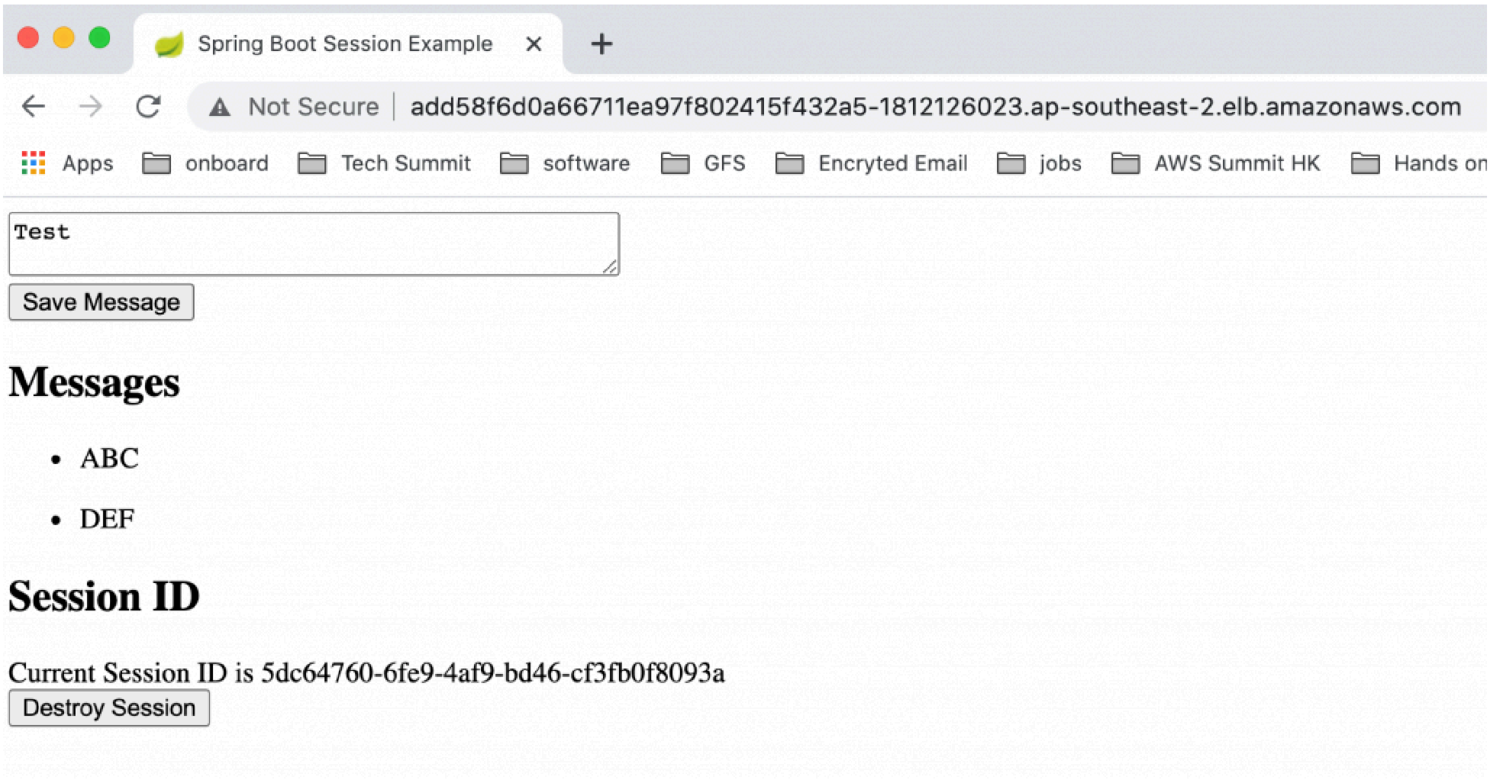
Let's also review the created service `kubectl get svc`.

```
NAME         TYPE           CLUSTER-IP      EXTERNAL-IP                                  PORT(S)           AGE
    SELECTOR

java-ms       LoadBalancer    172.20.83.176      ***-***.eu-central-1.elb.amazonaws.com       80:32300/TCP      33s
      app=java-ms

kubernetes    ClusterIP       172.20.0.1          <none>                                        443/TCP
 2d1h        <none>
```

What we can see here is that the Service with name java-ms has an External-IP assigned to it. This is the DNS Name of the Classic Loadbalancer that is created behind the scenes. If you open that URL, you should see the Website (this might take a few minutes for the ELB to be provisioned).

Testing and verification

The webpage that opens should look similar to the following screenshot. In the text field you can enter text that is saved on clicking the "**Save Message**" button. This text will be listed in the "**Messages**" as shown in the following screenshot. These messages are saved as session data and now persists at Amazon ElastiCache Redis.



Test

Messages

- ABC
- DEF

Session ID

Current Session ID is 5dc64760-6fe9-4af9-bd46-cf3fb0f8093a

By destroying the session, you will lose the saved messages.

Cleaning up

To avoid incurring future charges, you should delete all created resources after you are finished with testing. The repository contains a destroy.sh script. This script takes care to delete all deployed resources.

The script requires one parameter -p that requires the aws cli profile name that should be used:

```
./destroy.sh -p <profile name>
```

Conclusion

This post showed you the end-to-end setup of a stateful Java service running on Amazon EKS. The service is made scalable by saving the user sessions and the according session data in a Redis database. This solution requires changing the application code, and there are situations where this is not an option. By using StatefulSets as Kubernetes Resource in combination with an Application Load Balancer and sticky sessions, the goal of replicating the service can still be achieved.

We chose to use a Kubernetes Service in combination with a Classic Load Balancer. For a production workload, managing incoming traffic with a Kubernetes Ingress and an Application Load Balancer might be the better option. If you want to know more about Kubernetes Ingress with Amazon EKS, visit our Application Load Balancing on Amazon EKS documentation.

Field Notes provides hands-on technical guidance from AWS Solutions Architects, consultants, and technical account managers, based on their experiences in the field solving real-world business problems for customers.

TAGS: Field Notes