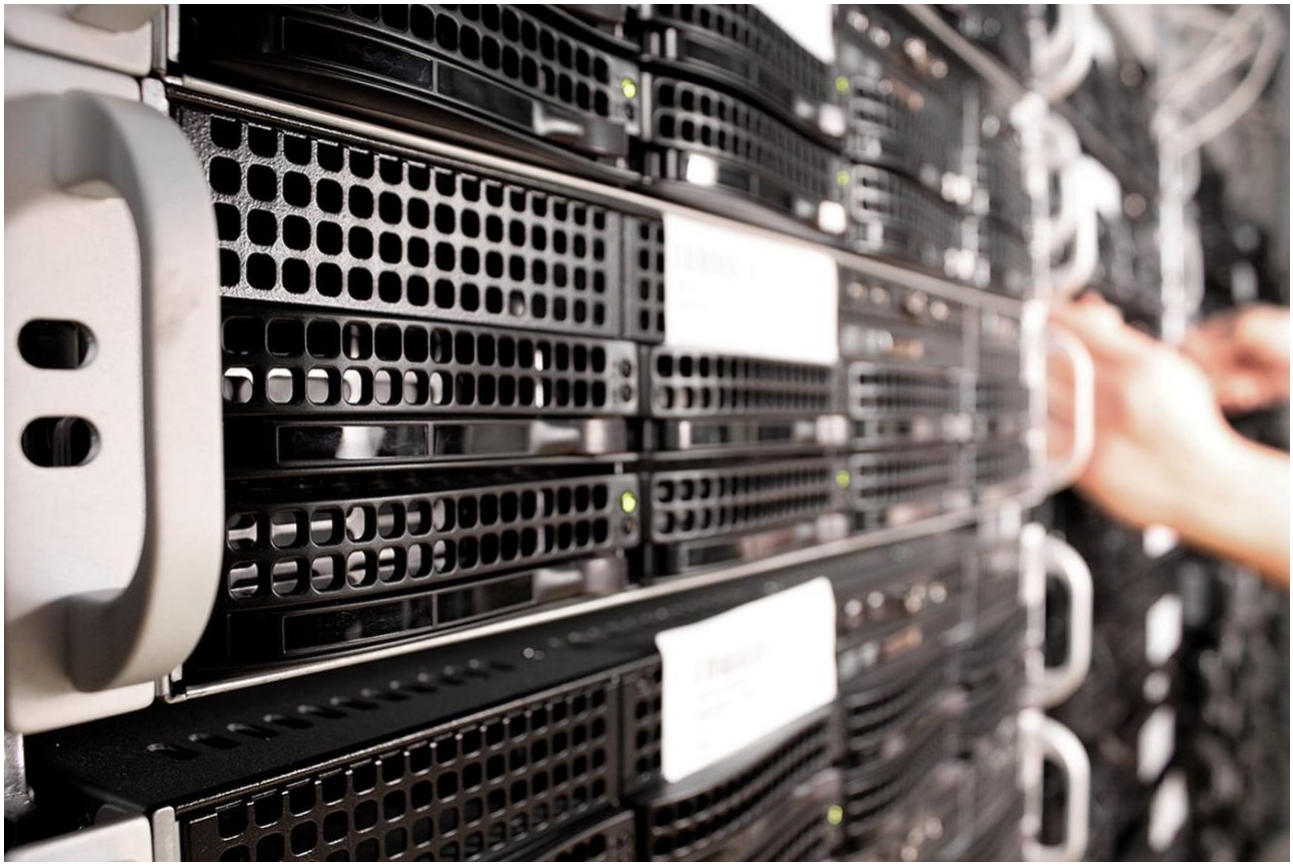
How to programmatically access Kubernetes cluster with aws-iam-authenticator

Miro Barsocchi



I need to access a Kubernetes (k8s) cluster and verify the version, status, or whatever of Pods inside the cluster. This can be done with **kubectl** binary given by k8s but also programmatically using client and SDK developed by **kubernetes** team.

kubectl client, which can be used from terminal/bash, has some simple configuration (it depends on your implementation). Mine was:

AWS ACCESS_KEY, SECRET, and REGION should be exported as environment variables, so before all, there should be something like this (this is valid for Apple. Windows and Linux have different syntax)

```
export AWS_ACCESS_KEY_ID=IOP342752ADE3456156
export AWS_SECRET_ACCESS_KEY=KN!/kFssdfFWewcASF12dasWE365574FFo
export AWS_DEFAULT_REGION=eu-west-1
```

Then there is a file used for configuration, named **.kubenv**, which paths should be exported as well

```
export KUBECONFIG=/home/mysuer/.kubenv
```

The file content is something like this (YML format)

```
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data:
4oCU4oCTQkVHSU4gQ0VSVEIGSUNBVEXigJTigJMKTUlJQy9qQ0NsMmdBd0lCQWdJTEJBQUFBQUFC
RlV0YXc1UXdEUVlKS29aSWh2Y05BUUVGQlFBd1Z6RUxNQWtHCllXeFRhV2R1SUc1MkxYTmhNUkF3R
GdZRFZRUUxFd2RTYjI5MElFTkJNUnN3R1FZRFZRUURFeEpIYkc5aVlXeFQKYVdkdUlGSnZiM1FnUTBF
d2dnRWlNQTBHQ1NxR1NJYjNEUUVCQVFVQUE0SUJEd0F3Z2dFS0FvSUJBUURhRHVhWgpqYzZqND
ArS2Z2dnhpNE1sYStwSUgvRXFzTG1WRVFTOThHUFI0bWRtenh6ZHp4dElLKzZOaVk2YXJ5bUFaYXZw
CjM4TmZsTlVWeVJSQm5NUmRkV1FWRGY5Vk1PeUdqLzhON3l5NVkwYjJxdnpmdkduOUxoSklaSnJnbG
ZDbTd5bVAKSE1VZnBJQnZGU0RKM2d5SUNoM1dabFhpL0VqSktTWnA0QT09CuKAlOKAk0VORCBDRV
```

```
JUSUZJQ0FUReKAlOKAkw==
    server: https://ABCDEFGHILMNOPQRSTUVZ1234567890.gr7.eu-west-1.eks.amazonaws.com
  name: arn:aws:eks:eu-west-1:111111111111:cluster/a-k8s-cluster-01
contexts:
- context:
    cluster: arn:aws:eks:eu-west-1:111111111111:cluster/a-k8s-cluster-01
    user: arn:aws:eks:eu-west-1:111111111111:cluster/a-k8s-cluster-01
  name: arn:aws:eks:eu-west-1:111111111111:cluster/a-k8s-cluster-01
current-context: arn:aws:eks:eu-west-1:111111111111:cluster/a-k8s-cluster-01
kind: Config
preferences: {}
users:
- name: arn:aws:eks:eu-west-1:111111111111:cluster/a-k8s-cluster-01
  user:
    exec:
      apiVersion: client.authentication.k8s.io/v1beta1
      args:
      - token
      - -i
      - a-k8s-cluster-01
      command: /var/bin/aws-iam-authenticator
      env:
      - name: "AWS_PROFILE"
        value: "this-is-a-k8s-clusters"
      interactiveMode: IfAvailable
      provideClusterInfo: false
```

After the two exports, you can easily use this command to get all the namespaces from Kubernetes cluster:

```
kubectl get namespaces
```

And, for each namespace, get information about PODS

```
kubectl get deployments -o wide -n A_NAMESPACE
```

...where A_NAMESPACE should be each namespace returned from the previous command.

How to do authentication and retrieve POD information from k8s (Kubernetes) cluster in Java

To do the same using a Java client, we should authenticate in the k8s (Kubernetes) cluster and follow some steps.

Export AWS secret, access keys, and regions

This part can not be skipped, so even in Java we should export this value right before running every code

```
export AWS_ACCESS_KEY_ID=IOP342752ADE3456156
export AWS_SECRET_ACCESS_KEY=KN!/kFssdfFWewcASF12dasWE365574FFo
export AWS_DEFAULT_REGION=eu-west-1
```

I want to emulate this Code Example for kubernetes Java client but with all the authentication prerequisites. I need some modification that results in this code snippet

```
Region myRegion = Regions.EU_WEST_1;
String clusterName = "a-k8s-cluster-01";
```

```
String clusterHost = "https://ABCDEFGHILMNOPQRSTUVZ1234567890.gr7.";
String certificate =
"4oCU4oCTQkVHSU4gQ0VSVElGSUNBVEXigJTigJMKTUlJQy9qQ0NsMmdBd0lCQWdJTEJBQUFBQUF
CRlV0YXc1UXdEUVlKS29aSWh2Y05BUUVGQlFBd1Z6RUxNQWtHCllXeFRhV2R1SUc1MkxYTmhNUkF3
RGdZRFZRUUxFd2RTYjI5MElFTkJNUnN3R1FZRFZRUURFeEpIYkc5aVlXeFQKYVdkdUlGSnZiM1FnUTB
Fd2dnRWlNQTBHQ1NxR1NJYjNEUUVCQVFVQUE0SUJEd0F3Z2dFS0FvSUJBUURhRHVhWgpqYzZqND
ArS2Z2dnhpNE1sYStwSUgvRXFzTG1WRVFTOThHUFI0bWRtenh6ZHp4dElLKzZOaVk2YXJ5bUFaYXZw
CjM4TmZsTlVWeVJSQm5NUmRkV1FWRGY5Vk1PeUdqLzhON3l5NVkwYjJxdnpmdkduOUxoSklaSnJnbG
ZDbTd5bVAKSE1VZnBJQnZGU0RKM2d5SUNoM1dabFhpL0VqSktTWnA0QT09CuKAlOKAk0VORCBDRV
JUSUZJQ0FUReKAlOKAkw==";

AWSCredentialsProvider credentialsProvider = new DefaultAWSCredentialsProviderChain();
AWSSecurityTokenServiceClient tokenService = (AWSSecurityTokenServiceClient)
AWSSecurityTokenServiceClientBuilder
        .standard()
        .withCredentials(credentialsProvider)
        .withRegion(myRegion)
        .build();
String initialToken = generateToken(clusterName,
        new Date(System.currentTimeMillis() + TimeUnit.SECONDS.toMillis(60)),
        "sts",
        myRegion.getName(),
        tokenService,
        credentialsProvider,
        "https",
        "sts." + myRegion.getName() + ".amazonaws.com");

byte[] bytes = Base64.getDecoder().decode(certificate.getBytes());
String endPoint = clusterHost + myRegion.getName()
        + ".eks.amazonaws.com";
client = (new ClientBuilder())
        .setBasePath(endPoint)
        .setAuthentication(new AccessTokenAuthentication(initialToken))
        .setVerifyingSsl(true)
        .setCertificateAuthority(bytes)
```

```
        .build();

Configuration.setDefaultApiClient(client);

CoreV1Api api = new CoreV1Api();
V1NamespaceList namespaces = api.listNamespace(null, null, null, null, null, -1, null, null, 30, false);
...
..
.
```

After this, having namespaces and api instantiated, everything is straightforward. Before analyzing how we arrive at this solution, let me add the snippet for the method **generateToken** that is a copy/paste from this thread. It works, no need to investigate further

```
private String generateToken(String clusterName,
                             Date expirationDate,
                             String serviceName,
                             String region,
                             AWSSecurityTokenServiceClient awsSecurityTokenServiceClient,
                             AWSCredentialsProvider credentialsProvider,
                             String scheme,
                             String host) throws URISyntaxException {
    try {
        DefaultRequest<GetCallerIdentityRequest> callerIdentityRequestDefaultRequest = new
DefaultRequest<>(new GetCallerIdentityRequest(), serviceName);
        URI uri = new URI(scheme, host, null, null);
        callerIdentityRequestDefaultRequest.setResourcePath("/");
        callerIdentityRequestDefaultRequest.setEndpoint(uri);
        callerIdentityRequestDefaultRequest.setHttpMethod(HttpMethodName.GET);
        callerIdentityRequestDefaultRequest.addParameter("Action", "GetCallerIdentity");
        callerIdentityRequestDefaultRequest.addParameter("Version", "2011-06-15");
```

```
        callerIdentityRequestDefaultRequest.addHeader("x-k8s-aws-id", clusterName);

        Signer signer = SignerFactory.createSigner(SignerFactory.VERSION_FOUR_SIGNER, new
SignerParams(serviceName, region));
        SignerProvider signerProvider = new DefaultSignerProvider(awsSecurityTokenServiceClient, signer);
        PresignerParams presignerParams = new PresignerParams(uri,
                credentialsProvider,
                signerProvider,
                SdkClock.STANDARD);

        PresignerFacade presignerFacade = new PresignerFacade(presignerParams);
        URL url = presignerFacade.presign(callerIdentityRequestDefaultRequest, expirationDate);
        String encodedUrl = Base64.getUrlEncoder().withoutPadding().encodeToString(url.toString().getBytes());
        log.info("Token [{}]", encodedUrl);
        return "k8s-aws-v1." + encodedUrl;
    } catch (URISyntaxException e) {
        log.error("could not generate token", e);
        throw e;
    }
}
```

The main part of the code is “how we move from **.kubenv** config file to a Java code solution”. The following variables (at the beginning of the java code)

```
Region myRegion = Regions.EU_WEST_1;
String clusterName = "a-k8s-cluster-01";
String clusterHost = "https://ABCDEFGHILMNOPQRSTUVZ1234567890.gr7.";
String certificate =
"4oCU4oCTQkVHSU4gQ0VSVElGSUNBVEXigJTigJMKTUlJQy9qQ0NsMmdBd0lCQWdJTEJBQUFBQUF
CRlV0YXc1UXdEUVlKS29aSWh2Y05BUUVGQlFBd1Z6RUxNQWtHCllXeFRhV2R1SUc1MkxYTmhNUkF3
RGdZRFZRUUxFd2RTYjI5MElFTkJNUnN3R1FZRFZRUURFeEpIYkc5aVlXeFQKYVdkdUlGSnZiM1FnUTB
```

```
Fd2dnRWlNQTBHQ1NxR1NJYjNEUUVCQVFVQUE0SUJEd0F3Z2dFS0FvSUJBUURhRHVhWgpqYzZqND
ArS2Z2dnhpNE1sYStwSUgvRXFzTG1WRVFTOThHUFI0bWRtenh6ZHp4dElLKzZOaVk2YXJ5bUFaYXZw
CjM4TmZsTlVWeVJSQm5NUmRkV1FWRGY5Vk1PeUdqLzhON3l5NVkwYjJxdnpmdkduOUxoSklaSnJnbG
ZDbTd5bVAKSE1VZnBJQnZGU0RKM2d5SUNoM1dabFhpL0VqSktTWnA0QT09CuKAlOKAk0VORCBDRV
JUSUZJQ0FURReKAlOKAkw==";
```

Are easy to explain:

myRegion: is a region that you may find in the **.kubenv** config file, in every server or host nodes there should be something like “eu-west-1”

clusterName: you may find it in your **.kubenv** config file, as the final part of the node hierarchy clusters-cluster-name or in the field ***users*** **->** ***user*** **->** ***exec*** **->** ***args***. It is used as an argument when we call the binary **aws-iam-authenticator**

clusterHost: in the kube config file **.kubenv** is the field ***cluster*** **->** ***cluster*** **->** ***server***

certificate: in the kube config file **.kubenv** is the field ***cluster*** **->** ***certificate-authority-data***. This represents a base64 encoded certificate, if you decode it with a Base6e decoder, the string result is:

```
—BEGIN CERTIFICATE—
MIIC/jCCl2gAwIBAgILBAAAAAABFUtaw5QwDQYJKoZIhvcNAQEFBQAwVzELMAkG
YWxTaWduIG52LXNhMRAwDgYDVQQLEwdSb290IENBMRswGQYDVQQDExJHbG9iYWxT
aWduIFJvb3QgQ0EwggEiMA0GCSqGSIb3DQEBAQUAA4IBDwAwggEKAoIBAQDaDuaZ
jc6j40+Kfvvxi4Mla+pIH/EqsLmVEQS98GPR4mdmzxzdzxtIK+6NiY6arymAZavp
38NflNUVyRRBnMRddWQVDf9VMOyGj/8N7yy5Y0b2qvzfvGn9LhJIZJrglfCm7ymP
```

```
HMUfpIBvFSDJ3gyICh3WZlXi/EjJKSZp4A==
-----END CERTIFICATE-----
```

Everything should work as expected and you should have access to all the API in Java.

Ref.

[Need aws-iam-authenticator thread]

[Code Example for kubernetes Java client]