Red Hat build of Keycloak 26.2

High Availability Guide

Last Updated: 2025-09-19

Legal Notice

Abstract

This guide consists of information for administrators to configure and use the Red Hat build of Keycloak 26.2 for high availability.

Table of Contents

CHAPTER 1. MULTI-SITE DEPLOYMENTS

Connect multiple Red Hat build of Keycloak deployments in different sites to increase the overall availability.

Red Hat build of Keycloak supports deployments that consist of multiple Red Hat build of Keycloak instances that connect to each other using its Infinispan caches; load balancers can distribute the load evenly across those instances. Those setups are intended for a transparent network on a single site.

The Red Hat build of Keycloak high-availability guide goes one step further to describe setups across multiple sites. While this setup adds additional complexity, that extra amount of high availability may be needed for some environments.

1.1. WHEN TO USE A MULTI-SITE SETUP

The multi-site deployment capabilities of Red Hat build of Keycloak are targeted at use cases that:

- Are constrained to a single AWS Region.
- Permit planned outages for maintenance.
- Fit within a defined user and request count.
- Can accept the impact of periodic outages.

1.2. SUPPORTED CONFIGURATION

- Two Openshift single-AZ clusters, in the same AWS Region
  - Provisioned with Red Hat OpenShift Service on AWS (ROSA), either ROSA HCP or ROSA classic.
  - Each Openshift cluster has all its workers in a single Availability Zone.
  - OpenShift version 4.17 (or later).
- Amazon Aurora PostgreSQL database
  - High availability with a primary DB instance in one Availability Zone, and a synchronously replicated reader in the second Availability Zone
  - Version 16.1
- AWS Global Accelerator, sending traffic to both ROSA clusters
- AWS Lambda to automate failover

Any deviation from the configuration above is not supported and any issue must be replicated in that environment for support.

Read more on each item in the Building blocks multi-site deployments chapter.

1.3. MAXIMUM LOAD

- 100,000 users

- 300 requests per second

See the Concepts for sizing CPU and memory resources chapter for more information.

1.4. LIMITATIONS

- During upgrades of Red Hat build of Keycloak or Data Grid both sites needs to be taken offline for the duration of the upgrade.
- During certain failure scenarios, there may be downtime of up to 5 minutes.
- After certain failure scenarios, manual intervention may be required to restore redundancy by bringing the failed site back online.
- During certain switchover scenarios, there may be downtime of up to 5 minutes.

For more details on limitations see the Concepts for multi-site deployments chapter.

1.5. NEXT STEPS

The different chapters introduce the necessary concepts and building blocks. For each building block, a blueprint shows how to set a fully functional example. Additional performance tuning and security hardening are still recommended when preparing a production setup.

CHAPTER 2. CONCEPTS FOR MULTI-SITE DEPLOYMENTS

Understand multi-site deployment with synchronous replication.

This topic describes a highly available multi-site setup and the behavior to expect. It outlines the requirements of the high availability architecture and describes the benefits and tradeoffs.

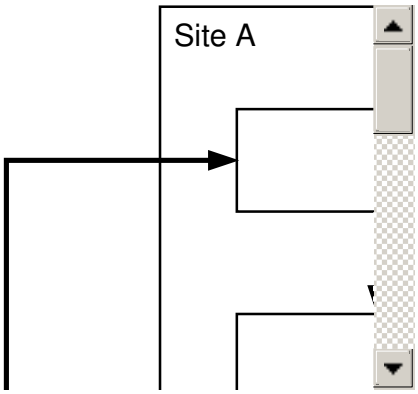
2.1. WHEN TO USE THIS SETUP

Use this setup to provide Red Hat build of Keycloak deployments that are able to tolerate site failures, reducing the likelihood of downtime.

2.2. DEPLOYMENT, DATA STORAGE AND CACHING

Two independent Red Hat build of Keycloak deployments running in different sites are connected with a low latency network connection. Users, realms, clients, sessions, and other entities are stored in a database that is replicated synchronously across the two sites. The data is also cached in the Red Hat build of Keycloak Infinispan caches as local caches. When the data is changed in one Red Hat build of Keycloak instance, that data is updated in the database, and an invalidation message is sent to the other site using the **work** cache.

In the following paragraphs and diagrams, references to deploying Data Grid apply to the external Data Grid.



2.3. CAUSES OF DATA AND SERVICE LOSS

While this setup aims for high availability, the following situations can still lead to service or data loss:

- Red Hat build of Keycloak site failure may result in requests failing in the period between the failure and the loadbalancer detecting it, as requests may still be routed to the failed site.
- Once failures occur in the communication between the sites, manual steps are necessary to re-synchronize a degraded setup.
- Degraded setups can lead to service or data loss if additional components fail. Monitoring is necessary to detect degraded setups.

2.4. FAILURES WHICH THIS SETUP CAN SURVIVE

Failure	Recovery	RPO[1]	RTO[2]
Database node	If the writer instance fails, the database can promote a reader instance in the same or other site to be the new writer.	No data loss	Seconds to minutes (depending on the database)
Red Hat build of Keycloak node	Multiple Red Hat build of Keycloak instances run on each site. If one instance fails some incoming requests might receive an error message or are delayed for some seconds.	No data loss	Less than 30 seconds
Data Grid node	Multiple Data Grid instances run in each site. If one instance fails, it takes a few seconds for the other nodes to notice the change. Entities are stored in at least two Data Grid nodes, so a single node failure does not lead to data loss.	No data loss	Less than 30 seconds
Data Grid cluster failure	If the Data Grid cluster fails in one of the sites, Red Hat build of Keycloak will not be able to communicate with the external Data Grid on that site, and the Red Hat build of Keycloak service will be unavailable. The loadbalancer will detect the situation as **/lb-check** returns an error, and will direct all traffic to the other site. The setup is degraded until the Data Grid cluster is restored and the data is re-synchronized.	No data loss[3]	Seconds to minutes (depending on load balancer setup)

Failure	Recovery	RPO[1]	RTO[2]
Connectivity Data Grid	If the connectivity between the two sites is lost, data cannot be sent to the other site. Incoming requests might receive an error message or are delayed for some seconds. The Data Grid will mark the other site offline, and will stop sending data. One of the sites needs to be taken offline in the loadbalancer until the connection is restored and the data is re-synchronized between the two sites. In the blueprints, we show how this can be automated.	No data loss[3]	Seconds to minutes (depending on load balancer setup)
Connectivity database	If the connectivity between the two sites is lost, the synchronous replication will fail. Some requests might receive an error message or be delayed for a few seconds. Manual operations might be necessary depending on the database.	No data loss[3]	Seconds to minutes (depending on the database)
Site failure	If none of the Red Hat build of Keycloak nodes are available, the loadbalancer will detect the outage and redirect the traffic to the other site. Some requests might receive an error message until the loadbalancer detects the failure.	No data loss[3]	Less than two minutes

Table footnotes:

[1] Recovery point objective, assuming all parts of the setup were healthy at the time this occurred.

[2] Recovery time objective.

[3] Manual operations needed to restore the degraded setup.

The statement “No data loss” depends on the setup not being degraded from previous failures, which includes completing any pending manual operations to resynchronize the state between the sites.

2.5. KNOWN LIMITATIONS

Site Failure

A successful failover requires a setup not degraded from previous failures. All manual operations like a re-synchronization after a previous failure must be complete to prevent data loss. Use monitoring to ensure degradations are detected and handled in a timely manner.

Out-of-sync sites

The sites can become out of sync when a synchronous Data Grid request fails. This situation is currently difficult to monitor, and it would need a full manual re-sync of Data Grid to recover. Monitoring the number of cache entries in both sites and the Red Hat build of Keycloak log file can show when resynch would become necessary.

Manual operations

Manual operations that re-synchronize the Data Grid state between the sites will issue a full state transfer which will put a stress on the system.

Two sites restriction

This setup is tested and supported only with two sites. Each additional site increases overall latency as it is necessary for data to be synchronously written to each site. Furthermore, the probability of network failures, and therefore downtime, also increases. Therefore, we do not support more than two sites as we believe it would lead to a deployment with inferior stability and performance.

2.6. QUESTIONS AND ANSWERS

Why synchronous database replication?

A synchronously replicated database ensures that data written in one site is always available in the other site after site failures and no data is lost. It also ensures that the next request will not return stale data, independent on which site it is served.

Why synchronous Data Grid replication?

A synchronously replicated Data Grid ensures that cached data in one site are always available on the other site after a site failure and no data is lost. It also ensures that the next request will not return stale data, independent on which site it is served.

Why is a low-latency network between sites needed?

Synchronous replication defers the response to the caller until the data is received at the other site. For synchronous database replication and synchronous Data Grid replication, a low latency is necessary as each request can have potentially multiple interactions between the sites when data is updated which would amplify the latency.

Is a synchronous cluster less stable than an asynchronous cluster?

An asynchronous setup would handle network failures between the sites gracefully, while the synchronous setup would delay requests and will throw errors to the caller where the asynchronous setup would have deferred the writes to Data Grid or the database on the other site. However, as the two sites would never be fully up-to-date, this setup could lead to data loss during failures. This would include:

- Lost changes leading to users being able to log in with an old password because database changes are not replicated to the other site at the point of failure when using an asynchronous database.

- Invalid caches leading to users being able to log in with an old password because invalidating caches are not propagated at the point of failure to the other site when using an asynchronous Data Grid replication.

Therefore, tradeoffs exist between high availability and consistency. The focus of this topic is to prioritize consistency over availability with Red Hat build of Keycloak.

2.7. NEXT STEPS

Continue reading in the Building blocks multi-site deployments chapter to find blueprints for the different building blocks.

CHAPTER 3. BUILDING BLOCKS MULTI-SITE DEPLOYMENTS

Learn about building blocks and suggested setups for multi-site deployments.

The following building blocks are needed to set up a multi-site deployment with synchronous replication.

The building blocks link to a blueprint with an example configuration. They are listed in the order in which they need to be installed.

NOTE

We provide these blueprints to show a minimal functionally complete example with a good baseline performance for regular installations. You would still need to adapt it to your environment and your organization's standards and security best practices.

3.1. PREREQUISITES

- Understanding the concepts laid out in the Concepts for multi-site deployments chapter.

3.2. TWO SITES WITH LOW-LATENCY CONNECTION

Ensures that synchronous replication is available for both the database and the external Data Grid.

Suggested setup: Two AWS Availability Zones within the same AWS Region.

Not considered: Two regions on the same or different continents, as it would increase the latency and the likelihood of network failures. Synchronous replication of databases as services with Aurora Regional Deployments on AWS is only available within the same region.

3.3. ENVIRONMENT FOR RED HAT BUILD OF KEYCLOAK AND DATA GRID

Ensures that the instances are deployed and restarted as needed.

Suggested setup: Red Hat OpenShift Service on AWS (ROSA) deployed in each availability zone.

Not considered: A stretched ROSA cluster which spans multiple availability zones, as this could be a single point of failure if misconfigured.

3.4. DATABASE

A synchronously replicated database across two sites.

Blueprint: Deploying AWS Aurora in multiple availability zones .

3.5. DATA GRID

A deployment of Data Grid that leverages the Data Grid's Cross-DC functionality.

Blueprint: Deploying Data Grid for HA with the Data Grid Operator using the Data Grid Operator, and connect the two sites using Data Grid's Gossip Router.

Not considered: Direct interconnections between the Kubernetes clusters on the network layer. It might be considered in the future.

3.6. RED HAT BUILD OF KEYCLOAK

A clustered deployment of Red Hat build of Keycloak in each site, connected to an external Data Grid.

Blueprint: Deploying Red Hat build of Keycloak for HA with the Operator that includes connecting to the Aurora database and the Data Grid server.

3.7. LOAD BALANCER

A load balancer which checks the **/lb-check** URL of the Red Hat build of Keycloak deployment in each site, plus an automation to detect Data Grid connectivity problems between the two sites.

Blueprint: Deploying an AWS Global Accelerator load balancer together with Deploying an AWS Lambda to disable a non-responding site.

CHAPTER 4. CONCEPTS FOR DATABASE CONNECTION POOLS

Understand concepts for avoiding resource exhaustion and congestion.

This section is intended when you want to understand considerations and best practices on how to configure database connection pools for Red Hat build of Keycloak. For a configuration where this is applied, visit Deploying Red Hat build of Keycloak for HA with the Operator .

4.1. CONCEPTS

Creating new database connections is expensive as it takes time. Creating them when a request arrives will delay the response, so it is good to have them created before the request arrives. It can also contribute to a stampede effect where creating a lot of connections in a short time makes things worse as it slows down the system and blocks threads. Closing a connection also invalidates all server side statements caching for that connection.

For the best performance, the values for the initial, minimal and maximum database connection pool size should all be equal. This avoids creating new database connections when a new request comes in which is costly.

Keeping the database connection open for as long as possible allows for server side statement caching bound to a connection. In the case of PostgreSQL, to use a server-side prepared statement, a query needs to be executed (by default) at least five times.

See the PostgreSQL docs on prepared statements for more information.

CHAPTER 5. CONCEPTS FOR CONFIGURING THREAD POOLS

Understand concepts for avoiding resource exhaustion and congestion.

This section is intended when you want to understand the considerations and best practices on how to configure thread pools connection pools for Red Hat build of Keycloak. For a configuration where this is applied, visit Deploying Red Hat build of Keycloak for HA with the Operator .

5.1. CONCEPTS

5.1.1. JGroups communications

JGroups communications, which is used in single-site setups for the communication between Red Hat build of Keycloak nodes, benefits from the use of virtual threads which are available in OpenJDK 21 when at least two cores are available for Red Hat build of Keycloak. This reduces the memory usage and removes the need to configure thread pool sizes. Therefore, the use of OpenJDK 21 is recommended.

5.1.2. Quarkus executor pool

Red Hat build of Keycloak requests, as well as blocking probes, are handled by an executor pool. Depending on the available CPU cores, it has a maximum size of 50 or more threads. Threads are created as needed, and will end when no longer needed, so the system will scale up and down automatically. Red Hat build of Keycloak allows configuring the maximum thread pool size by the **http-pool-max-threads** configuration option. See Deploying Red Hat build of Keycloak for HA with the Operator for an example.

When running on Kubernetes, adjust the number of worker threads to avoid creating more load than what the CPU limit allows for the Pod to avoid throttling, which would lead to congestion. When running on physical machines, adjust the number of worker threads to avoid creating more load than the node can handle to avoid congestion. Congestion would result in longer response times and an increased memory usage, and eventually an unstable system.

Ideally, you should start with a low limit of threads and adjust it accordingly to the target throughput and response time. When the load and the number of threads increases, the database connections can also become a bottleneck. Once a request cannot acquire a database connection within 5 seconds, it will fail with a message in the log like **Unable to acquire JDBC Connection**. The caller will receive a response with a 5xx HTTP status code indicating a server side error.

If you increase the number of database connections and the number of threads too much, the system will be congested under a high load with requests queueing up, which leads to a bad performance. The number of database connections is configured via the **Database** settings **db-pool-initial-size**, **db-pool-min-size** and **db-pool-max-size** respectively. Low numbers ensure fast response times for all clients, even if there is an occasionally failing request when there is a load spike.

5.1.3. Load Shedding

By default, Red Hat build of Keycloak will queue all incoming requests infinitely, even if the request processing stalls. This will use additional memory in the Pod, can exhaust resources in the load balancers, and the requests will eventually time out on the client side without the client knowing if the request has been processed. To limit the number of queued requests in Red Hat build of Keycloak, set an additional Quarkus configuration option.

Configure **http-max-queued-requests** to specify a maximum queue length to allow for effective load shedding once this queue size is exceeded. Assuming a Red Hat build of Keycloak Pod processes around 200 requests per second, a queue of 1000 would lead to maximum waiting times of around 5 seconds.

When this setting is active, requests that exceed the number of queued requests will return with an HTTP 503 error. Red Hat build of Keycloak logs the error message in its log.

5.1.4. Probes

Red Hat build of Keycloak's liveness probe is non-blocking to avoid a restart of a Pod under a high load.

The overall health probe and the readiness probe can in some cases block to check the connection to the database, so they might fail under a high load. Due to this, a Pod can become non-ready under a high load.

5.1.5. OS Resources

In order for Java to create threads, when running on Linux it needs to have file handles available. Therefore, the number of open files (as retrieved as **ulimit -n** on Linux) need to provide head-space for Red Hat build of Keycloak to increase the number of threads needed. Each thread will also consume memory, and the container memory limits need to be set to a value that allows for this or the Pod will be killed by Kubernetes.

CHAPTER 6. CONCEPTS FOR SIZING CPU AND MEMORY RESOURCES

Understand concepts for avoiding resource exhaustion and congestion.

Use this as a starting point to size a product environment. Adjust the values for your environment as needed based on your load tests.

6.1. PERFORMANCE RECOMMENDATIONS

WARNING

- Performance will be lowered when scaling to more Pods (due to additional overhead) and using a cross-datacenter setup (due to additional traffic and operations).
- Increased cache sizes can improve the performance when Red Hat build of Keycloak instances running for a longer time. This will decrease response times and reduce IOPS on the database. Still, those caches need to be filled when an instance is restarted, so do not set resources too tight based on the stable state measured once the caches have been filled.
- Use these values as a starting point and perform your own load tests before going into production.

Summary:

- The used CPU scales linearly with the number of requests up to the tested limit below.

Recommendations:

- The base memory usage for a Pod including caches of Realm data and 10,000 cached sessions is 1250 MB of RAM.
- In containers, Keycloak allocates 70% of the memory limit for heap-based memory. It will also use approximately 300 MB of non-heap-based memory. To calculate the requested memory, use the calculation above. As memory limit, subtract the non-heap memory from the value above and divide the result by 0.7.
- For each 15 password-based user logins per second, allocate 1 vCPU to the cluster (tested with up to 300 per second).
  Red Hat build of Keycloak spends most of the CPU time hashing the password provided by the user, and it is proportional to the number of hash iterations.
- For each 120 client credential grants per second, 1 vCPU to the cluster (tested with up to 2000 per second).*
  Most CPU time goes into creating new TLS connections, as each client runs only a single request.

- For each 120 refresh token requests per second, 1 vCPU to the cluster (tested with up to 435 refresh token requests per second).*

- Leave 150% extra head-room for CPU usage to handle spikes in the load. This ensures a fast startup of the node, and enough capacity to handle failover tasks. Performance of Red Hat build of Keycloak dropped significantly when its Pods were throttled in our tests.

- When performing requests with more than 2500 different clients concurrently, not all client information will fit into Red Hat build of Keycloak's caches when those are using the standard cache sizes of 10000 entries each. Due to this, the database may become a bottleneck as client data is reloaded frequently from the database. To reduce the database usage, increase the **users** cache size by two times the number of concurrently used clients, and the **realms** cache size by four times the number of concurrently used clients.

Red Hat build of Keycloak, which by default stores user sessions in the database, requires the following resources for optimal performance on an Aurora PostgreSQL multi-AZ database:

For every 100 login/logout/refresh requests per second:

- Budget for 1400 Write IOPS.

- Allocate between 0.35 and 0.7 vCPU.

The vCPU requirement is given as a range, as with an increased CPU saturation on the database host the CPU usage per request decreases while the response times increase. A lower CPU quota on the database can lead to slower response times during peak loads. Choose a larger CPU quota if fast response times during peak loads are critical. See below for an example.

6.1.1. Measuring the activity of a running Red Hat build of Keycloak instance

Sizing of a Red Hat build of Keycloak instance depends on the actual and forecasted numbers for password-based user logins, refresh token requests, and client credential grants as described in the previous section.

To retrieve the actual numbers of a running Red Hat build of Keycloak instance for these three key inputs, use the metrics Red Hat build of Keycloak provides:

- The user event metric **keycloak_user_events_total** for event type **login** includes both password-based logins and cookie-based logins, still it can serve as a first approximate input for this sizing guide.

- To find out number of password validations performed by Red Hat build of Keycloak use the metric **keycloak_credentials_password_hashing_validations_total**. The metric also contains tags providing some details about the hashing algorithm used and the outcome of the validation. Here is the list of available tags: **realm**, **algorithm**, **hashing_strength**, **outcome**.

- Use the user event metric **keycloak_user_events_total** for the event types **refresh_token** and **client_login** for refresh token requests and client credential grants respectively.

See the {links_observability_event-metrics_name} and {links_observability_metrics-for-troubleshooting-http_name} chapters for more information.

These metrics are crucial for tracking daily and weekly fluctuations in user activity loads, identifying emerging trends that may indicate the need to resize the system and validating sizing calculations. By systematically measuring and evaluating these user event metrics, you can ensure your system remains appropriately scaled and responsive to changes in user behavior and demand.

6.1.2. Calculation example (single site)

Target size:

- 45 logins and logouts per seconds
- 360 client credential grants per second*
- 360 refresh token requests per second (1:8 ratio for logins)*
- 3 Pods

Limits calculated:

- CPU requested per Pod: 3 vCPU
  (45 logins per second = 3 vCPU, 360 client credential grants per second = 3 vCPU, 360 refresh tokens = 3 vCPU. This sums up to 9 vCPU total. With 3 Pods running in the cluster, each Pod then requests 3 vCPU)
- CPU limit per Pod: 7.5 vCPU
  (Allow for an additional 150% CPU requested to handle peaks, startups and failover tasks)
- Memory requested per Pod: 1250 MB
  (1250 MB base memory)
- Memory limit per Pod: 1360 MB
  (1250 MB expected memory usage minus 300 non-heap-usage, divided by 0.7)
- Aurora Database instance: either **db.t4g.large** or **db.t4g.xlarge** depending on the required response times during peak loads.
  (45 logins per second, 5 logouts per second, 360 refresh tokens per seconds. This sums up to 410 requests per second. This expected DB usage is 1.4 to 2.8 vCPU, with a DB idle load of 0.3 vCPU. This indicates either a 2 vCPU **db.t4g.large** instance or a 4 vCPU **db.t4g.xlarge** instance. A 2 vCPU **db.t4g.large** would be more cost-effective if the response times are allowed to be higher during peak usage. In our tests, the median response time for a login and a token refresh increased by up to 120 ms once the CPU saturation reached 90% on a 2 vCPU **db.t4g.large** instance given this scenario. For faster response times during peak usage, consider a 4 vCPU **db.t4g.xlarge** instance for this scenario.)

6.1.3. Sizing a multi-site setup

To create the sizing an active-active Keycloak setup with two AZs in one AWS region, following these steps:

- Create the same number of Pods with the same memory sizing as above on the second site.
- The database sizing remains unchanged. Both sites will connect to the same database writer instance.

In regard to the sizing of CPU requests and limits, there are different approaches depending on the expected failover behavior:

Fast failover and more expensive

Keep the CPU requests and limits as above for the second site. This way any remaining site can take over the traffic from the primary site immediately without the need to scale.

Slower failover and more cost-effective

Reduce the CPU requests and limits as above by 50% for the second site. When one of the sites fails, scale the remaining site from 3 Pod to 6 Pods either manually, automated, or using a Horizontal Pod Autoscaler. This requires enough spare capacity on the cluster or cluster auto-scaling capabilities.

Alternative setup for some environments

Reduce the CPU requests by 50% for the second site, but keep the CPU limits as above. This way, the remaining site can take the traffic, but only at the downside that the Nodes will experience CPU pressure and therefore slower response times during peak traffic. The benefit of this setup is that the number of Pods does not need to scale during failovers which is simpler to set up.

6.2. REFERENCE ARCHITECTURE

The following setup was used to retrieve the settings above to run tests of about 10 minutes for different scenarios:

- OpenShift 4.17.x deployed on AWS via ROSA.
- Machine pool with **c7g.2xlarge** instances.*
- Red Hat build of Keycloak deployed with the Operator and 3 pods in a high-availability setup with two sites in active/active mode.
- OpenShift's reverse proxy runs in the passthrough mode where the TLS connection of the client is terminated at the Pod.
- Database Amazon Aurora PostgreSQL in a multi-AZ setup.
- Default user password hashing with Argon2 and 5 hash iterations and minimum memory size 7 MiB as recommended by OWASP (which is the default).
- Client credential grants do not use refresh tokens (which is the default).
- Database seeded with 20,000 users and 20,000 clients.
- Infinispan local caches at default of 10,000 entries, so not all clients and users fit into the cache, and some requests will need to fetch the data from the database.
- All authentication sessions in distributed caches as per default, with two owners per entries, allowing one failing Pod without losing data.
- All user and client sessions are stored in the database and are not cached in-memory as this was tested in a multi-site setup. Expect a slightly higher performance for single-site setups as a fixed number of user and client sessions will be cached.
- OpenJDK 21

* For non-ARM CPU architectures on AWS (**c7i**/**c7a** vs. **c7g**) we found that client credential grants and refresh token workloads were able to deliver up to two times the number of operations per CPU core, while password hashing was delivering a constant number of operations per CPU core. Depending on your workload and your cloud pricing, please run your own tests and make your own calculations for mixed workloads to find out which architecture delivers a better pricing for you.

CHAPTER 7. CONCEPTS TO AUTOMATE DATA GRID CLI COMMANDS

Data Grid CLI commands can be automated by creating a `Batch` CR instance.

When interacting with an external Data Grid in Kubernetes, the **Batch** CR allows you to automate this using standard **oc** commands.

7.1. WHEN TO USE IT

Use this when automating interactions on Kubernetes. This avoids providing usernames and passwords and checking shell script outputs and their status.

For human interactions, the CLI shell might still be a better fit.

7.2. EXAMPLE

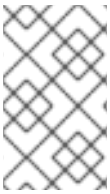
The following **Batch** CR takes a site offline as described in the operational procedure Taking a site offline.

```
apiVersion: infinispan.org/v2alpha1
kind: Batch
metadata:
  name: take-offline
  namespace: keycloak (1)
spec:
  cluster: infinispan (2)
  config: | (3)
    site take-offline --all-caches --site=site-a
    site status --all-caches --site=site-a
```

1. The **Batch** CR must be created in the same namespace as the Data Grid deployment.
2. The name of the Infinispan CR.
3. A multiline string containing one or more Data Grid CLI commands.

Once the CR has been created, wait for the status to show the completion.

```
oc -n keycloak wait --for=jsonpath='{.status.phase}'=Succeeded Batch/take-offline
```



NOTE

Modifying a **Batch** CR instance has no effect. Batch operations are "one-time" events that modify Infinispan resources. To update **.spec** fields for the CR, or when a batch operation fails, you must create a new instance of the **Batch** CR.

7.3. FURTHER READING

For more information, see the Data Grid Operator **Batch** CR documentation.

CHAPTER 8. DEPLOYING AWS AURORA IN MULTIPLE AVAILABILITY ZONES

Deploy an AWS Aurora as the database building block in a multi-site deployment.

This topic describes how to deploy an Aurora regional deployment of a PostgreSQL instance across multiple availability zones to tolerate one or more availability zone failures in a given AWS region.

This deployment is intended to be used with the setup described in the Concepts for multi-site deployments chapter. Use this deployment with the other building blocks outlined in the Building blocks multi-site deployments chapter.

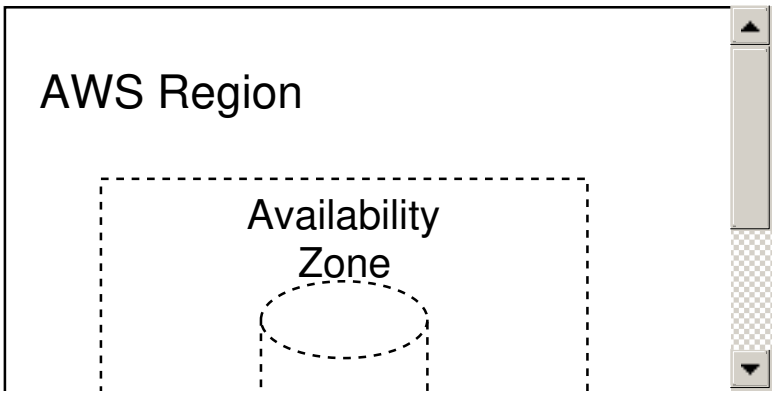
NOTE

We provide these blueprints to show a minimal functionally complete example with a good baseline performance for regular installations. You would still need to adapt it to your environment and your organization's standards and security best practices.

8.1. ARCHITECTURE

Aurora database clusters consist of multiple Aurora database instances, with one instance designated as the primary writer and all others as backup readers. To ensure high availability in the event of availability zone failures, Aurora allows database instances to be deployed across multiple zones in a single AWS region. In the event of a failure on the availability zone that is hosting the Primary database instance, Aurora automatically heals itself and promotes a reader instance from a non-failed availability zone to be the new writer instance.

Figure 8.1. Aurora Multiple Availability Zone Deployment



See the AWS Aurora documentation for more details on the semantics provided by Aurora databases.

This documentation follows AWS best practices and creates a private Aurora database that is not exposed to the Internet. To access the database from a ROSA cluster, establish a peering connection between the database and the ROSA cluster.

8.2. PROCEDURE

The following procedure contains two sections:

- Creation of an Aurora Multi-AZ database cluster with the name "keycloak-aurora" in eu-west-1.
- Creation of a peering connection between the ROSA cluster(s) and the Aurora VPC to allow applications deployed on the ROSA clusters to establish connections with the database.

8.2.1. Create Aurora database Cluster

1. Create a VPC for the Aurora cluster

Command:

```
aws ec2 create-vpc \
  --cidr-block 192.168.0.0/16 \
  --tag-specifications "ResourceType=vpc, Tags=[{Key=AuroraCluster,Value=keycloak-aurora}]" \ 1
  --region eu-west-1
```

1 We add an optional tag with the name of the Aurora cluster so that we can easily retrieve the VPC.

Output:

```
{
    "Vpc": {
        "CidrBlock": "192.168.0.0/16",
        "DhcpOptionsId": "dopt-0bae7798158bc344f",
        "State": "pending",
        "VpcId": "vpc-0b40bd7c59dbe4277",
        "OwnerId": "606671647913",
        "InstanceTenancy": "default",
        "Ipv6CidrBlockAssociationSet": [],
        "CidrBlockAssociationSet": [
            {
                "AssociationId": "vpc-cidr-assoc-09a02a83059ba5ab6",
                "CidrBlock": "192.168.0.0/16",
                "CidrBlockState": {
                    "State": "associated"
                }
            }
        ],
        "IsDefault": false
    }
}
```

2. Create a subnet for each availability zone that Aurora will be deployed to, using the **VpcId** of the newly created VPC.

NOTE

The cidr-block range specified for each of the availability zones must not overlap.

a. Zone A

Command:

```
aws ec2 create-subnet \
  --availability-zone "eu-west-1a" \
  --vpc-id vpc-0b40bd7c59dbe4277 \
```

```
  --cidr-block 192.168.0.0/19 \
  --region eu-west-1
```

Output:


```
{
    "Subnet": {
        "AvailabilityZone": "eu-west-1a",
        "AvailabilityZoneId": "euw1-az3",
        "AvailableIpAddressCount": 8187,
        "CidrBlock": "192.168.0.0/19",
        "DefaultForAz": false,
        "MapPublicIpOnLaunch": false,
        "State": "available",
        "SubnetId": "subnet-0d491a1a798aa878d",
        "VpcId": "vpc-0b40bd7c59dbe4277",
        "OwnerId": "606671647913",
        "AssignIpv6AddressOnCreation": false,
        "Ipv6CidrBlockAssociationSet": [],
        "SubnetArn": "arn:aws:ec2:eu-west-1:606671647913:subnet/subnet-
0d491a1a798aa878d",
        "EnableDns64": false,
        "Ipv6Native": false,
        "PrivateDnsNameOptionsOnLaunch": {
            "HostnameType": "ip-name",
            "EnableResourceNameDnsARecord": false,
            "EnableResourceNameDnsAAAARecord": false
        }
    }
}
```


b. Zone B

Command:

```
aws ec2 create-subnet \
  --availability-zone "eu-west-1b" \
  --vpc-id vpc-0b40bd7c59dbe4277 \
  --cidr-block 192.168.32.0/19 \
  --region eu-west-1
```

Output:


```
{
    "Subnet": {
        "AvailabilityZone": "eu-west-1b",
        "AvailabilityZoneId": "euw1-az1",
        "AvailableIpAddressCount": 8187,
        "CidrBlock": "192.168.32.0/19",
        "DefaultForAz": false,
        "MapPublicIpOnLaunch": false,
        "State": "available",
        "SubnetId": "subnet-057181b1e3728530e",
        "VpcId": "vpc-0b40bd7c59dbe4277",
        "OwnerId": "606671647913",
```

```
        "AssignIpv6AddressOnCreation": false,
        "Ipv6CidrBlockAssociationSet": [],
        "SubnetArn": "arn:aws:ec2:eu-west-1:606671647913:subnet/subnet-
    057181b1e3728530e",
        "EnableDns64": false,
        "Ipv6Native": false,
        "PrivateDnsNameOptionsOnLaunch": {
            "HostnameType": "ip-name",
            "EnableResourceNameDnsARecord": false,
            "EnableResourceNameDnsAAAARecord": false
        }
    }
}
```


3. Obtain the ID of the Aurora VPC route-table

Command:

```
aws ec2 describe-route-tables \
  --filters Name=vpc-id,Values=vpc-0b40bd7c59dbe4277 \
  --region eu-west-1
```

Output:


```
{
    "RouteTables": [
        {
            "Associations": [
                {
                    "Main": true,
                    "RouteTableAssociationId": "rtbassoc-02dfa06f4c7b4f99a",
                    "RouteTableId": "rtb-04a644ad3cd7de351",
                    "AssociationState": {
                        "State": "associated"
                    }
                }
            ],
            "PropagatingVgws": [],
            "RouteTableId": "rtb-04a644ad3cd7de351",
            "Routes": [
                {
                    "DestinationCidrBlock": "192.168.0.0/16",
                    "GatewayId": "local",
                    "Origin": "CreateRouteTable",
                    "State": "active"
                }
            ],
            "Tags": [],
            "VpcId": "vpc-0b40bd7c59dbe4277",
            "OwnerId": "606671647913"
        }
    ]
}
```


4. Associate the Aurora VPC route-table each availability zone's subnet

a. Zone A

Command:

```
aws ec2 associate-route-table \
  --route-table-id rtb-04a644ad3cd7de351 \
  --subnet-id subnet-0d491a1a798aa878d \
  --region eu-west-1
```

b. Zone B

Command:

```
aws ec2 associate-route-table \
  --route-table-id rtb-04a644ad3cd7de351 \
  --subnet-id subnet-057181b1e3728530e \
  --region eu-west-1
```

5. Create Aurora Subnet Group

Command:

```
aws rds create-db-subnet-group \
  --db-subnet-group-name keycloak-aurora-subnet-group \
  --db-subnet-group-description "Aurora DB Subnet Group" \
  --subnet-ids subnet-0d491a1a798aa878d subnet-057181b1e3728530e \
  --region eu-west-1
```

6. Create Aurora Security Group

Command:

```
aws ec2 create-security-group \
  --group-name keycloak-aurora-security-group \
  --description "Aurora DB Security Group" \
  --vpc-id vpc-0b40bd7c59dbe4277 \
  --region eu-west-1
```

Output:

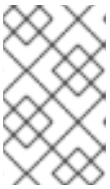
```
{
    "GroupId": "sg-0d746cc8ad8d2e63b"
}
```

7. Create the Aurora DB Cluster

Command:

```
aws rds create-db-cluster \
    --db-cluster-identifier keycloak-aurora \
    --database-name keycloak \
    --engine aurora-postgresql \
    --engine-version ${properties["aurora-postgresql.version"]} \
```

```
--master-username keycloak \
--master-user-password secret99 \
--vpc-security-group-ids sg-0d746cc8ad8d2e63b \
--db-subnet-group-name keycloak-aurora-subnet-group \
--region eu-west-1
```



NOTE

You should replace the **--master-username** and **--master-user-password** values. The values specified here must be used when configuring the Red Hat build of Keycloak database credentials.

Output:


```
{
    "DBCluster": {
        "AllocatedStorage": 1,
        "AvailabilityZones": [
            "eu-west-1b",
            "eu-west-1c",
            "eu-west-1a"
        ],
        "BackupRetentionPeriod": 1,
        "DatabaseName": "keycloak",
        "DBClusterIdentifier": "keycloak-aurora",
        "DBClusterParameterGroup": "default.aurora-postgresql15",
        "DBSubnetGroup": "keycloak-aurora-subnet-group",
        "Status": "creating",
        "Endpoint": "keycloak-aurora.cluster-clhthfqe0h8p.eu-west-1.rds.amazonaws.com",
        "ReaderEndpoint": "keycloak-aurora.cluster-ro-clhthfqe0h8p.eu-west-
1.rds.amazonaws.com",
        "MultiAZ": false,
        "Engine": "aurora-postgresql",
        "EngineVersion": "15.5",
        "Port": 5432,
        "MasterUsername": "keycloak",
        "PreferredBackupWindow": "02:21-02:51",
        "PreferredMaintenanceWindow": "fri:03:34-fri:04:04",
        "ReadReplicaIdentifiers": [],
        "DBClusterMembers": [],
        "VpcSecurityGroups": [
            {
                "VpcSecurityGroupId": "sg-0d746cc8ad8d2e63b",
                "Status": "active"
            }
        ],
        "HostedZoneId": "Z29XKXDKYMONMX",
        "StorageEncrypted": false,
        "DbClusterResourceId": "cluster-IBWXUWQYM3MS5BH557ZJ6ZQU4I",
        "DBClusterArn": "arn:aws:rds:eu-west-1:606671647913:cluster:keycloak-aurora",
        "AssociatedRoles": [],
        "IAMDatabaseAuthenticationEnabled": false,
        "ClusterCreateTime": "2023-11-01T10:40:45.964000+00:00",
        "EngineMode": "provisioned",
        "DeletionProtection": false,
```

```
        "HttpEndpointEnabled": false,
        "CopyTagsToSnapshot": false,
        "CrossAccountClone": false,
        "DomainMemberships": [],
        "TagList": [],
        "AutoMinorVersionUpgrade": true,
        "NetworkType": "IPV4"
    }
}
```

8. Create Aurora DB instances

   a. Create Zone A Writer instance

   Command:

   ```
   aws rds create-db-instance \
     --no-auto-minor-version-upgrade \
     --db-cluster-identifier keycloak-aurora \
     --db-instance-identifier "keycloak-aurora-instance-1" \
     --db-instance-class db.t4g.large \
     --engine aurora-postgresql \
     --region eu-west-1
   ```

   b. Create Zone B Reader instance

   Command:

   ```
   aws rds create-db-instance \
     --no-auto-minor-version-upgrade \
     --db-cluster-identifier keycloak-aurora \
     --db-instance-identifier "keycloak-aurora-instance-2" \
     --db-instance-class db.t4g.large \
     --engine aurora-postgresql \
     --region eu-west-1
   ```

9. Wait for all Writer and Reader instances to be ready

   Command:

   ```
   aws rds wait db-instance-available --db-instance-identifier keycloak-aurora-instance-1 --
   region eu-west-1
   aws rds wait db-instance-available --db-instance-identifier keycloak-aurora-instance-2 --
   region eu-west-1
   ```

10. Obtain the Writer endpoint URL for use by Keycloak

    Command:

    ```
    aws rds describe-db-clusters \
      --db-cluster-identifier keycloak-aurora \
      --query 'DBClusters[*].Endpoint' \
      --region eu-west-1 \
      --output text
    ```

Output:

```
[
    "keycloak-aurora.cluster-clhthfqe0h8p.eu-west-1.rds.amazonaws.com"
]
```

8.2.2. Establish Peering Connections with ROSA clusters

Perform these steps once for each ROSA cluster that contains a Red Hat build of Keycloak deployment.

1. Retrieve the Aurora VPC

   Command:

   ```
   aws ec2 describe-vpcs \
     --filters "Name=tag:AuroraCluster,Values=keycloak-aurora" \
     --query 'Vpcs[*].VpcId' \
     --region eu-west-1 \
     --output text
   ```

   Output:

   ```
   vpc-0b40bd7c59dbe4277
   ```

2. Retrieve the ROSA cluster VPC

   a. Log in to the ROSA cluster using **oc**

   b. Retrieve the ROSA VPC

      Command:

      ```
      NODE=$(oc get nodes --selector=node-role.kubernetes.io/worker -o
      jsonpath='{.items[0].metadata.name}')
      aws ec2 describe-instances \
        --filters "Name=private-dns-name,Values=${NODE}" \
        --query 'Reservations[0].Instances[0].VpcId' \
        --region eu-west-1 \
        --output text
      ```

      Output:

      ```
      vpc-0b721449398429559
      ```

3. Create Peering Connection

   Command:

   ```
   aws ec2 create-vpc-peering-connection \
     --vpc-id vpc-0b721449398429559 \ 1
     --peer-vpc-id vpc-0b40bd7c59dbe4277 \ 2
   ```

```
    --peer-region eu-west-1 \
    --region eu-west-1
```

1. ROSA cluster VPC
2. Aurora VPC

Output:


```
{
    "VpcPeeringConnection": {
        "AccepterVpcInfo": {
            "OwnerId": "606671647913",
            "VpcId": "vpc-0b40bd7c59dbe4277",
            "Region": "eu-west-1"
        },
        "ExpirationTime": "2023-11-08T13:26:30+00:00",
        "RequesterVpcInfo": {
            "CidrBlock": "10.0.17.0/24",
            "CidrBlockSet": [
                {
                    "CidrBlock": "10.0.17.0/24"
                }
            ],
            "OwnerId": "606671647913",
            "PeeringOptions": {
                "AllowDnsResolutionFromRemoteVpc": false,
                "AllowEgressFromLocalClassicLinkToRemoteVpc": false,
                "AllowEgressFromLocalVpcToRemoteClassicLink": false
            },
            "VpcId": "vpc-0b721449398429559",
            "Region": "eu-west-1"
        },
        "Status": {
            "Code": "initiating-request",
            "Message": "Initiating Request to 606671647913"
        },
        "Tags": [],
        "VpcPeeringConnectionId": "pcx-0cb23d66dea3dca9f"
    }
}
```


4. Wait for Peering connection to exist

   Command:

   ```
   aws ec2 wait vpc-peering-connection-exists --vpc-peering-connection-ids pcx-0cb23d66dea3dca9f
   ```

5. Accept the peering connection

   Command:

```
aws ec2 accept-vpc-peering-connection \
  --vpc-peering-connection-id pcx-0cb23d66dea3dca9f \
  --region eu-west-1
```

Output:


```
{
    "VpcPeeringConnection": {
        "AccepterVpcInfo": {
            "CidrBlock": "192.168.0.0/16",
            "CidrBlockSet": [
                {
                    "CidrBlock": "192.168.0.0/16"
                }
            ],
            "OwnerId": "606671647913",
            "PeeringOptions": {
                "AllowDnsResolutionFromRemoteVpc": false,
                "AllowEgressFromLocalClassicLinkToRemoteVpc": false,
                "AllowEgressFromLocalVpcToRemoteClassicLink": false
            },
            "VpcId": "vpc-0b40bd7c59dbe4277",
            "Region": "eu-west-1"
        },
        "RequesterVpcInfo": {
            "CidrBlock": "10.0.17.0/24",
            "CidrBlockSet": [
                {
                    "CidrBlock": "10.0.17.0/24"
                }
            ],
            "OwnerId": "606671647913",
            "PeeringOptions": {
                "AllowDnsResolutionFromRemoteVpc": false,
                "AllowEgressFromLocalClassicLinkToRemoteVpc": false,
                "AllowEgressFromLocalVpcToRemoteClassicLink": false
            },
            "VpcId": "vpc-0b721449398429559",
            "Region": "eu-west-1"
        },
        "Status": {
            "Code": "provisioning",
            "Message": "Provisioning"
        },
        "Tags": [],
        "VpcPeeringConnectionId": "pcx-0cb23d66dea3dca9f"
    }
}
```


6. Update ROSA cluster VPC route-table

Command:

```
ROSA_PUBLIC_ROUTE_TABLE_ID=$(aws ec2 describe-route-tables \
  --filters "Name=vpc-id,Values=vpc-0b721449398429559"
```

```
"Name=association.main,Values=true" \ 1
  --query "RouteTables[*].RouteTableId" \
  --output text \
  --region eu-west-1
)
aws ec2 create-route \
  --route-table-id ${ROSA_PUBLIC_ROUTE_TABLE_ID} \
  --destination-cidr-block 192.168.0.0/16 \ 2
  --vpc-peering-connection-id pcx-0cb23d66dea3dca9f \
  --region eu-west-1
```

1 ROSA cluster VPC

2 This must be the same as the cidr-block used when creating the Aurora VPC

7. Update the Aurora Security Group

Command:

```
AURORA_SECURITY_GROUP_ID=$(aws ec2 describe-security-groups \
  --filters "Name=group-name,Values=keycloak-aurora-security-group" \
  --query "SecurityGroups[*].GroupId" \
  --region eu-west-1 \
  --output text
)
aws ec2 authorize-security-group-ingress \
  --group-id ${AURORA_SECURITY_GROUP_ID} \
  --protocol tcp \
  --port 5432 \
  --cidr 10.0.17.0/24 \ 1
  --region eu-west-1
```

1 The "machine_cidr" of the ROSA cluster

Output:

```
{
    "Return": true,
    "SecurityGroupRules": [
        {
            "SecurityGroupRuleId": "sgr-0785d2f04b9cec3f5",
            "GroupId": "sg-0d746cc8ad8d2e63b",
            "GroupOwnerId": "606671647913",
            "IsEgress": false,
            "IpProtocol": "tcp",
            "FromPort": 5432,
            "ToPort": 5432,
            "CidrIpv4": "10.0.17.0/24"
        }
    ]
}
```

8.3. VERIFYING THE CONNECTION

The simplest way to verify that a connection is possible between a ROSA cluster and an Aurora DB cluster is to deploy **psql** on the Openshift cluster and attempt to connect to the writer endpoint.

The following command creates a pod in the default namespace and establishes a **psql** connection with the Aurora cluster if possible. Upon exiting the pod shell, the pod is deleted.

```
USER=keycloak 1
PASSWORD=secret99 2
DATABASE=keycloak 3
HOST=$(aws rds describe-db-clusters \
  --db-cluster-identifier keycloak-aurora \ 4
  --query 'DBClusters[*].Endpoint' \
  --region eu-west-1 \
  --output text
)
oc run -i --tty --rm debug --image=postgres:15 --restart=Never -- psql
postgresql://${USER}:${PASSWORD}@${HOST}/${DATABASE}
```

1. Aurora DB user, this can be the same as **--master-username** used when creating the DB.
2. Aurora DB user-password, this can be the same as **--master—user-password** used when creating the DB.
3. The name of the Aurora DB, such as **--database-name**.
4. The name of your Aurora DB cluster.

8.4. CONNECTING AURORA DATABASE WITH RED HAT BUILD OF KEYCLOAK

Now that an Aurora database has been established and linked with all of your ROSA clusters, here are the relevant Red Hat build of Keycloak CR options to connect the Aurora database with Red Hat build of Keycloak. These changes will be required in the Deploying Red Hat build of Keycloak for HA with the Operator chapter. The JDBC url is configured to use the Aurora database writer endpoint.

1. Update **spec.db.url** to be **jdbc:aws-wrapper:postgresql://$HOST:5432/keycloak** where **$HOST** is the Aurora writer endpoint URL.
2. Ensure that the Secrets referenced by **spec.db.usernameSecret** and **spec.db.passwordSecret** contain usernames and passwords defined when creating Aurora.

8.5. NEXT STEPS

After successful deployment of the Aurora database continue with Deploying Data Grid for HA with the Data Grid Operator

CHAPTER 9. DEPLOYING DATA GRID FOR HA WITH THE DATA GRID OPERATOR

Deploy Data Grid for high availability in multi availability zones on Kubernetes.

This chapter describes the procedures required to deploy Data Grid in a multiple-cluster environment (cross-site). For simplicity, this topic uses the minimum configuration possible that allows Red Hat build of Keycloak to be used with an external Data Grid.

This chapter assumes two OpenShift clusters named **Site-A** and **Site-B**.

This is a building block following the concepts described in the Concepts for multi-site deployments chapter. See the Multi-site deployments chapter for an overview.

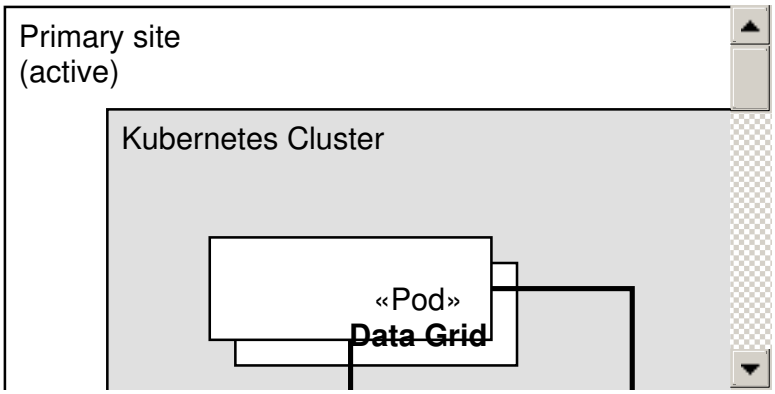
IMPORTANT

Only Data Grid version 8.5.3 or more recent patch releases are supported for external Data Grid deployments.

9.1. ARCHITECTURE

This setup deploys two synchronously replicating Data Grid clusters in two sites with a low-latency network connection. An example of this scenario could be two availability zones in one AWS region.

Red Hat build of Keycloak, loadbalancer and database have been removed from the following diagram for simplicity.



9.2. PREREQUISITES

- OpenShift or Kubernetes cluster running
- Understanding of the Data Grid Operator

9.3. PROCEDURE

1. Install the Data Grid Operator
2. Configure the credential to access the Data Grid cluster.
   Red Hat build of Keycloak needs this credential to be able to authenticate with the Data Grid cluster. The following **identities.yaml** file sets the username and password with admin permissions

   -

```
credentials:
 - username: developer
   password: strong-password
   roles:
     - admin
```

The **identities.yaml** could be set in a secret as one of the following:

- As a Kubernetes Resource:

  Credential Secret

```
apiVersion: v1
kind: Secret
type: Opaque
metadata:
  name: connect-secret
  namespace: keycloak
data:
  identities.yaml:
Y3JlZGVudGlhbHM6CiAgLSB1c2VybmFtZTogZGV2ZWxvcGVyCiAgICBwYXNzd29yZDog
c3Ryb25nLXBhc3N3b3JkCiAgICByb2xlczoKICAgICAgLSBhZG1pbgo= (1)
```

  (1) The **identities.yaml** from the previous example base64 encoded.

- Using the CLI

```
oc create secret generic connect-secret --from-file=identities.yaml
```

  Check the Configuring Authentication documentation for more details.

  These commands must be executed on both OpenShift clusters.

3. Create a service account.
   A service account is required to establish a connection between clusters. The Data Grid Operator uses it to inspect the network configuration from the remote site and to configure the local Data Grid cluster accordingly.

   For more details, see the Managing Cross-Site Connections documentation.

   a. Create a **service-account-token** secret type as follows. The same YAML file can be used in both OpenShift clusters.

   xsite-sa-secret-token.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: ispn-xsite-sa-token (1)
  annotations:
    kubernetes.io/service-account.name: "xsite-sa" (2)
type: kubernetes.io/service-account-token
```

1. The secret name.
2. The service account name.

b. Create the service account and generate an access token in both OpenShift clusters.

Create the service account in Site-A

```
oc create sa -n keycloak xsite-sa
oc policy add-role-to-user view -n keycloak -z xsite-sa
oc create -f xsite-sa-secret-token.yaml
oc get secrets ispn-xsite-sa-token -o jsonpath="{.data.token}" | base64 -d > Site-A-token.txt
```

Create the service account in Site-B

```
oc create sa -n keycloak xsite-sa
oc policy add-role-to-user view -n keycloak -z xsite-sa
oc create -f xsite-sa-secret-token.yaml
oc get secrets ispn-xsite-sa-token -o jsonpath="{.data.token}" | base64 -d > Site-B-token.txt
```

c. The next step is to deploy the token from **Site-A** into **Site-B** and the reverse:

Deploy Site-B token into Site-A

```
oc create secret generic -n keycloak xsite-token-secret \
  --from-literal=token="$(cat Site-B-token.txt)"
```

Deploy Site-A token into Site-B

```
oc create secret generic -n keycloak xsite-token-secret \
  --from-literal=token="$(cat Site-A-token.txt)"
```

4. Create TLS secrets
In this chapter, Data Grid uses an OpenShift Route for the cross-site communication. It uses the SNI extension of TLS to direct the traffic to the correct Pods. To achieve that, JGroups use TLS sockets, which require a Keystore and Truststore with the correct certificates.

For more information, see the Securing Cross Site Connections documentation or this Red Hat Developer Guide.

Upload the Keystore and the Truststore in an OpenShift Secret. The secret contains the file content, the password to access it, and the type of the store. Instructions for creating the certificates and the stores are beyond the scope of this chapter.

To upload the Keystore as a Secret, use the following command:

Deploy a Keystore

```
oc -n keycloak create secret generic xsite-keystore-secret \
  --from-file=keystore.p12="./certs/keystore.p12" \ 1
```

```
    --from-literal=password=secret \ 2
    --from-literal=type=pkcs12 3
```

1. The filename and the path to the Keystore.
2. The password to access the Keystore.
3. The Keystore type.

To upload the Truststore as a Secret, use the following command:

Deploy a Truststore

```
oc -n keycloak create secret generic xsite-truststore-secret \
      --from-file=truststore.p12="./certs/truststore.p12" \ 1
      --from-literal=password=caSecret \ 2
      --from-literal=type=pkcs12 3
```

1. The filename and the path to the Truststore.
2. The password to access the Truststore.
3. The Truststore type.

NOTE

Keystore and Truststore must be uploaded in both OpenShift clusters.

5. Create a Cluster for Data Grid with Cross-Site enabled
   The Setting Up Cross-Site documentation provides all the information on how to create and configure your Data Grid cluster with cross-site enabled, including the previous steps.

   A basic example is provided in this chapter using the credentials, tokens, and TLS Keystore/Truststore created by the commands from the previous steps.

The Infinispan CR for Site-A

```
apiVersion: infinispan.org/v1
kind: Infinispan
metadata:
  name: infinispan 1
  namespace: keycloak
  annotations:
    infinispan.org/monitoring: 'true' 2
spec:
  replicas: 3
  jmx:
    enabled: true
  security:
    endpointSecretName: connect-secret 3
  service:
    type: DataGrid
```

```
sites:
  local:
    name: site-a 4
    expose:
      type: Route 5
    maxRelayNodes: 128
    encryption:
      transportKeyStore:
        secretName: xsite-keystore-secret 6
        alias: xsite 7
        filename: keystore.p12 8
      routerKeyStore:
        secretName: xsite-keystore-secret 9
        alias: xsite 10
        filename: keystore.p12 11
      trustStore:
        secretName: xsite-truststore-secret 12
        filename: truststore.p12 13
  locations:
    - name: site-b 14
      clusterName: infinispan
      namespace: keycloak 15
      url: openshift://api.site-b 16
      secretName: xsite-token-secret 17
```

1. The cluster name
2. Allows the cluster to be monitored by Prometheus.
3. If using a custom credential, configure here the secret name.
4. The name of the local site, in this case **Site-A**.
5. Exposing the cross-site connection using OpenShift Route.
6. 9 The secret name where the Keystore exists as defined in the previous step.
7. 10 The alias of the certificate inside the Keystore.
8. 11 The secret key (filename) of the Keystore as defined in the previous step.
12. The secret name where the Truststore exists as defined in the previous step.
13. The Truststore key (filename) of the Keystore as defined in the previous step.
14. The remote site's name, in this case **Site-B**.
15. The namespace of the Data Grid cluster from the remote site.
16. The OpenShift API URL for the remote site.
17. The secret with the access token to authenticate into the remote site.

For **Site-B**, the **Infinispan** CR looks similar to the above. Note the differences in point 4, 11 and 13.

The Infinispan CR for Site-B

```
apiVersion: infinispan.org/v1
kind: Infinispan
metadata:
  name: infinispan (1)
  namespace: keycloak
  annotations:
    infinispan.org/monitoring: 'true' (2)
spec:
  replicas: 3
  jmx:
    enabled: true
  security:
    endpointSecretName: connect-secret (3)
  service:
    type: DataGrid
    sites:
      local:
        name: site-b (4)
        expose:
          type: Route (5)
        maxRelayNodes: 128
        encryption:
          transportKeyStore:
            secretName: xsite-keystore-secret (6)
            alias: xsite (7)
            filename: keystore.p12 (8)
          routerKeyStore:
            secretName: xsite-keystore-secret (9)
            alias: xsite (10)
            filename: keystore.p12 (11)
          trustStore:
            secretName: xsite-truststore-secret (12)
            filename: truststore.p12 (13)
      locations:
        - name: site-a (14)
          clusterName: infinispan
          namespace: keycloak (15)
          url: openshift://api.site-a (16)
          secretName: xsite-token-secret (17)
```

6. Creating the caches for Red Hat build of Keycloak.
   Red Hat build of Keycloak requires the following caches to be present: **actionTokens**, **authenticationSessions**, **loginFailures**, and **work**.

   The Data Grid Cache CR allows deploying the caches in the Data Grid cluster. Cross-site needs to be enabled per cache as documented by Cross Site Documentation. The documentation contains more details about the options used by this chapter. The following example shows the **Cache** CR for **Site-A**.

1. In **Site-A** create a **Cache** CR for each of the caches mentioned above with the following content.

Cache actionTokens

```
apiVersion: infinispan.org/v2alpha1
kind: Cache
metadata:
  name: actiontokens
  namespace: keycloak
spec:
  clusterName: infinispan
  name: actionTokens
  template: |-
    distributedCache:
      mode: "SYNC"
      owners: "2"
      statistics: "true"
      remoteTimeout: "5000"
      encoding:
        media-type: "application/x-protostream"
      locking:
        acquireTimeout: "4000"
      transaction:
        mode: "NON_DURABLE_XA" (1)
        locking: "PESSIMISTIC" (2)
      stateTransfer:
        chunkSize: "16"
      backups:
        site-b: (3)
          backup:
            strategy: "SYNC" (4)
            timeout: "4500" (5)
            failurePolicy: "FAIL" (6)
            stateTransfer:
              chunkSize: "16"
```

Cache authenticationSessions

```
apiVersion: infinispan.org/v2alpha1
kind: Cache
metadata:
  name: authenticationsessions
  namespace: keycloak
spec:
  clusterName: infinispan
  name: authenticationSessions
  template: |-
    distributedCache:
      mode: "SYNC"
      owners: "2"
      statistics: "true"
      remoteTimeout: "5000"
      encoding:
```

```
    media-type: "application/x-protostream"
  locking:
    acquireTimeout: "4000"
  transaction:
    mode: "NON_DURABLE_XA" (1)
    locking: "PESSIMISTIC" (2)
  stateTransfer:
    chunkSize: "16"
  indexing:
    enabled: true
    indexed-entities:
    - keycloak.RootAuthenticationSessionEntity
  backups:
    site-b: (3)
      backup:
        strategy: "SYNC" (4)
        timeout: "4500" (5)
        failurePolicy: "FAIL" (6)
        stateTransfer:
          chunkSize: "16"
```

Cache loginFailures

```
apiVersion: infinispan.org/v2alpha1
kind: Cache
metadata:
  name: loginfailures
  namespace: keycloak
spec:
  clusterName: infinispan
  name: loginFailures
  template: |-
    distributedCache:
      mode: "SYNC"
      owners: "2"
      statistics: "true"
      remoteTimeout: "5000"
      encoding:
        media-type: "application/x-protostream"
      locking:
        acquireTimeout: "4000"
      transaction:
        mode: "NON_DURABLE_XA" (1)
        locking: "PESSIMISTIC" (2)
      stateTransfer:
        chunkSize: "16"
      indexing:
        enabled: true
        indexed-entities:
        - keycloak.LoginFailureEntity
      backups:
        site-b: (3)
          backup:
            strategy: "SYNC" (4)
```

```
      timeout: "4500" 5
      failurePolicy: "FAIL" 6
      stateTransfer:
        chunkSize: "16"
```

Cache work

```
apiVersion: infinispan.org/v2alpha1
kind: Cache
metadata:
  name: work
  namespace: keycloak
spec:
  clusterName: infinispan
  name: work
  template: |-
    distributedCache:
      mode: "SYNC"
      owners: "2"
      statistics: "true"
      remoteTimeout: "5000"
      encoding:
        media-type: "application/x-protostream"
      locking:
        acquireTimeout: "4000"
      transaction:
        mode: "NON_DURABLE_XA" 1
        locking: "PESSIMISTIC" 2
      stateTransfer:
        chunkSize: "16"
      backups:
        site-b: 3
          backup:
            strategy: "SYNC" 4
            timeout: "4500" 5
            failurePolicy: "FAIL" 6
            stateTransfer:
              chunkSize: "16"
```

1 1 1 1 1 The transaction mode.

2 2 2 2 2 The locking mode used by the transaction.

3 3 3 3 3 The remote site name.

4 4 4 4 4 The cross-site communication strategy, in this case, **SYNC**.

5 5 5 5 5 The cross-site replication timeout.

6 9 6 6 6 6 The cross-site replication failure policy.

The example above is the recommended configuration to achieve the best data consistency.

Background information

Deadlocks may occur in an active-active setup as entries are modified concurrently in both sites.

The **transaction.mode: NON_DURABLE_XA** ensures that the transaction is rolled back keeping the data consistent if this occurs. The setting **backup.failurePolicy: FAIL** is required in this case. It will throw an error that allows the transaction to be safely rolled back. When this occurs, Red Hat build of Keycloak will attempt a retry.

The **transaction.locking: PESSIMISTIC** is the only supported locking mode; **OPTIMISTIC** is not recommended due to its network costs. The same settings also prevent that one site is updated while the other site is unreachable.

The **backup.strategy: SYNC** ensures the data is visible and stored in the other site when the Red Hat build of Keycloak request is completed.

NOTE

The **locking.acquireTimeout** can be reduced to fail fast in a deadlock scenario. The **backup.timeout** must always be higher than the **locking.acquireTimeout**.

For **Site-B**, the **Cache** CR is similar, except for the **backups.<name>** outlined in point 3 of the above diagram.

Example for actionTokens cache in Site-B

```
apiVersion: infinispan.org/v2alpha1
kind: Cache
metadata:
  name: actiontokens
  namespace: keycloak
spec:
  clusterName: infinispan
  name: actionTokens
  template: |-
    distributedCache:
      mode: "SYNC"
      owners: "2"
      statistics: "true"
      remoteTimeout: "5000"
      encoding:
        media-type: "application/x-protostream"
      locking:
        acquireTimeout: "4000"
      transaction:
        mode: "NON_DURABLE_XA" (1)
        locking: "PESSIMISTIC" (2)
      stateTransfer:
        chunkSize: "16"
      backups:
        site-a: (3)
          backup:
            strategy: "SYNC" (4)
            timeout: "4500" (5)
```

```
      failurePolicy: "FAIL" 6
      stateTransfer:
        chunkSize: "16"
```

9.4. VERIFYING THE DEPLOYMENT

Confirm that the Data Grid cluster is formed, and the cross-site connection is established between the OpenShift clusters.

Wait until the Data Grid cluster is formed

```
oc wait --for condition=WellFormed --timeout=300s infinispans.infinispan.org -n keycloak infinispan
```

Wait until the Data Grid cross-site connection is established

```
oc wait --for condition=CrossSiteViewFormed --timeout=300s infinispans.infinispan.org -n keycloak infinispan
```

9.5. CONNECTING DATA GRID WITH RED HAT BUILD OF KEYCLOAK

Now that the Data Grid server is running, here are the relevant Red Hat build of Keycloak CR changes necessary to connect it to Red Hat build of Keycloak. These changes will be required in the Deploying Red Hat build of Keycloak for HA with the Operator chapter.

1. Create a Secret with the username and password to connect to the external Data Grid deployment:

```
apiVersion: v1
kind: Secret
metadata:
  name: remote-store-secret
  namespace: keycloak
type: Opaque
data:
  username: ZGV2ZWxvcGVy # base64 encoding for 'developer'
  password: c2VjdXJlX3Bhc3N3b3Jk # base64 encoding for 'secure_password'
```

2. Extend the Red Hat build of Keycloak Custom Resource with **additionalOptions** as shown below.

NOTE

All the memory, resource and database configurations are skipped from the CR below as they have been described in the Deploying Red Hat build of Keycloak for HA with the Operator chapter already. Administrators should leave those configurations untouched.

```
apiVersion: k8s.keycloak.org/v2alpha1
kind: Keycloak
metadata:
  labels:
    app: keycloak
```

```
  name: keycloak
  namespace: keycloak
spec:
  additionalOptions:
    - name: cache-remote-host (1)
      value: "infinispan.keycloak.svc"
    - name: cache-remote-port (2)
      value: "11222"
    - name: cache-remote-username (3)
      secret:
        name: remote-store-secret
        key: username
    - name: cache-remote-password (4)
      secret:
        name: remote-store-secret
        key: password
    - name: db-driver
```

(1) The hostname of the remote Data Grid cluster.

(2) The port of the remote Data Grid cluster. This is optional and it defaults to **11222**.

(3) The Secret **name** and **key** with the Data Grid username credential.

(4) The Secret **name** and **key** with the Data Grid password credential.

(5) The **spi-connections-infinispan-quarkus-site-name** is an arbitrary Data Grid site name which Red Hat build of Keycloak needs for its Infinispan caches deployment when a remote store is used. This site-name is related only to the Infinispan caches and does not need to match any value from the external Data Grid deployment. If you are using multiple sites for Red Hat build of Keycloak in a cross-DC setup such as Deploying Data Grid for HA with the Data Grid Operator, the site name must be different in each site.

9.5.1. Architecture

This connects Red Hat build of Keycloak to Data Grid using TCP connections secured by TLS 1.3. It uses the Red Hat build of Keycloak's truststore to verify Data Grid's server certificate. As Red Hat build of Keycloak is deployed using its Operator on OpenShift in the prerequisites listed below, the Operator already added the **service-ca.crt** to the truststore which is used to sign Data Grid's server certificates. In other environments, add the necessary certificates to Red Hat build of Keycloak's truststore.

9.6. NEXT STEPS

After the Aurora AWS database and Data Grid are deployed and running, use the procedure in the Deploying Red Hat build of Keycloak for HA with the Operator chapter to deploy Red Hat build of Keycloak and connect it to all previously created building blocks.

9.7. RELEVANT OPTIONS

	Value
cache-remote-host The hostname of the external Infinispan cluster. Available only when feature **multi-site**, **clusterless** or **cache-embedded-remote-store** is set. CLI: **--cache-remote-host** Env: **KC_CACHE_REMOTE_HOST**	
cache-remote-password The password for the authentication to the external Infinispan cluster. It is optional if connecting to an unsecure external Infinispan cluster. If the option is specified, **cache-remote-username** is required as well. CLI: **--cache-remote-password** Env: **KC_CACHE_REMOTE_PASSWORD** Available only when remote host is set	
cache-remote-port The port of the external Infinispan cluster. CLI: **--cache-remote-port** Env: **KC_CACHE_REMOTE_PORT** Available only when remote host is set	**11222** (default)
cache-remote-tls-enabled Enable TLS support to communicate with a secured remote Infinispan server. Recommended to be enabled in production. CLI: **--cache-remote-tls-enabled** Env: **KC_CACHE_REMOTE_TLS_ENABLED** Available only when remote host is set	**true** (default), **false**
cache-remote-username The username for the authentication to the external Infinispan cluster. It is optional if connecting to an unsecure external Infinispan cluster. If the option is specified, **cache-remote-password** is required as well. CLI: **--cache-remote-username** Env: **KC_CACHE_REMOTE_USERNAME** Available only when remote host is set	

CHAPTER 10. DEPLOYING RED HAT BUILD OF KEYCLOAK FOR HA WITH THE OPERATOR

Deploy Red Hat build of Keycloak for high availability with the Red Hat build of Keycloak Operator as a building block.

This guide describes advanced Red Hat build of Keycloak configurations for Kubernetes which are load tested and will recover from single Pod failures.

These instructions are intended for use with the setup described in the Concepts for multi-site deployments chapter. Use it together with the other building blocks outlined in the Building blocks multi-site deployments chapter.

10.1. PREREQUISITES

- OpenShift or Kubernetes cluster running.
- Understanding of a Basic Red Hat build of Keycloak deployment of Red Hat build of Keycloak with the Red Hat build of Keycloak Operator.
- Aurora AWS database deployed using the Deploying AWS Aurora in multiple availability zones chapter.
- Data Grid server deployed using the Deploying Data Grid for HA with the Data Grid Operator chapter.
- Running Red Hat build of Keycloak with OpenJDK 21, which is the default for the containers distributed for Red Hat build of Keycloak, as this enabled virtual threads for the JGroups communication.

10.2. PROCEDURE

1. Determine the sizing of the deployment using the Concepts for sizing CPU and memory resources chapter.
2. Install the Red Hat build of Keycloak Operator as described in the Red Hat build of Keycloak Operator installation chapter.
3. Notice the configuration file below contains options relevant for connecting to the Aurora database from Deploying AWS Aurora in multiple availability zones
4. Notice the configuration file below options relevant for connecting to the Data Grid server from Deploying Data Grid for HA with the Data Grid Operator
5. Build a custom Red Hat build of Keycloak image which is prepared for usage with the Amazon Aurora PostgreSQL database.
6. Deploy the Red Hat build of Keycloak CR with the following values with the resource requests and limits calculated in the first step:

```
apiVersion: k8s.keycloak.org/v2alpha1
kind: Keycloak
metadata:
  labels:
    app: keycloak
```

```
  name: keycloak
  namespace: keycloak
spec:
  hostname:
    hostname: <KEYCLOAK_URL_HERE>
  resources:
    requests:
      cpu: "2"
      memory: "1250M"
    limits:
      cpu: "6"
      memory: "2250M"
  db:
    vendor: postgres
    url: jdbc:aws-wrapper:postgresql://<AWS_AURORA_URL_HERE>:5432/keycloak
    poolMinSize: 30 (1)
    poolInitialSize: 30
    poolMaxSize: 30
    usernameSecret:
      name: keycloak-db-secret
      key: username
    passwordSecret:
      name: keycloak-db-secret
      key: password
  image: <KEYCLOAK_IMAGE_HERE> (2)
  startOptimized: false (3)
  features:
    enabled:
      - multi-site (4)
  additionalOptions:
    - name: log-console-output
      value: json
    - name: metrics-enabled (5)
      value: 'true'
    - name: event-metrics-user-enabled
      value: 'true'
    - name: cache-remote-host
      value: "infinispan.keycloak.svc"
    - name: cache-remote-port
      value: "11222"
    - name: cache-remote-username
      secret:
        name: remote-store-secret
        key: username
    - name: cache-remote-password
      secret:
        name: remote-store-secret
        key: password
    - name: db-driver
      value: software.amazon.jdbc.Driver
  http:
    tlsSecret: keycloak-tls-secret
  instances: 3
```

1. The database connection pool initial, max and min size should be identical to allow statement caching for the database. Adjust this number to meet the needs of your system.
2. 3. Specify the URL to your custom Red Hat build of Keycloak image. If your image is optimized, set the **startOptimized** flag to **true**.
4. Enable additional features for multi-site support like the loadbalancer probe **/lb-check**.
5. To be able to analyze the system under load, enable the metrics endpoint.

10.3. VERIFYING THE DEPLOYMENT

Confirm that the Red Hat build of Keycloak deployment is ready.

```
oc wait --for=condition=Ready keycloaks.k8s.keycloak.org/keycloak
oc wait --for=condition=RollingUpdate=False keycloaks.k8s.keycloak.org/keycloak
```

10.4. OPTIONAL: LOAD SHEDDING

To enable load shedding, limit the number of queued requests.

Load shedding with max queued http requests

```
spec:
  additionalOptions:
    - name: http-max-queued-requests
      value: "1000"
```

All exceeding requests are served with an HTTP 503.

You might consider limiting the value for **http-pool-max-threads** further because multiple concurrent threads will lead to throttling by Kubernetes once the requested CPU limit is reached.

See the Concepts for configuring thread pools chapter about load shedding for details.

10.5. OPTIONAL: DISABLE STICKY SESSIONS

When running on OpenShift and the default passthrough Ingress setup as provided by the Red Hat build of Keycloak Operator, the load balancing done by HAProxy is done by using sticky sessions based on the IP address of the source. When running load tests, or when having a reverse proxy in front of HAProxy, you might want to disable this setup to avoid receiving all requests on a single Red Hat build of Keycloak Pod.

Add the following supplementary configuration under the **spec** in the Red Hat build of Keycloak Custom Resource to disable sticky sessions.

```
spec:
  ingress:
    enabled: true
    annotations:
      # When running load tests, disable sticky sessions on the OpenShift HAProxy router
```

```
    # to avoid receiving all requests on a single Red Hat build of Keycloak Pod.
    haproxy.router.openshift.io/balance: roundrobin
    haproxy.router.openshift.io/disable_cookies: 'true'
```

CHAPTER 11. DEPLOYING AN AWS GLOBAL ACCELERATOR LOAD BALANCER

Deploy an AWS Global Accelerator as the load-balancer building block in a multi-site deployment.

This topic describes the procedure required to deploy an AWS Global Accelerator to route traffic between multi-site Red Hat build of Keycloak deployments.

This deployment is intended to be used with the setup described in the Concepts for multi-site deployments chapter. Use this deployment with the other building blocks outlined in the Building blocks multi-site deployments chapter.

NOTE

We provide these blueprints to show a minimal functionally complete example with a good baseline performance for regular installations. You would still need to adapt it to your environment and your organization's standards and security best practices.

11.1. AUDIENCE

This chapter describes how to deploy an AWS Global Accelerator instance to handle Red Hat build of Keycloak client connection failover for multiple availability-zone Red Hat build of Keycloak deployments.

11.2. ARCHITECTURE

To ensure user requests are routed to each Red Hat build of Keycloak site we need to utilise a load balancer. To prevent issues with DNS caching on the client-side, the implementation should use a static IP address that remains the same when routing clients to both availability-zones.

In this chapter we describe how to route all Red Hat build of Keycloak client requests via an AWS Global Accelerator load balancer. In the event of a Red Hat build of Keycloak site failing, the Accelerator ensures that all client requests are routed to the remaining healthy site. If both sites are marked as unhealthy, then the Accelerator will "fail-open" and forward requests to a site chosen at random.

Figure 11.1. AWS Global Accelerator Failover

An AWS Network Load Balancer (NLB) is created on both ROSA clusters in order to make the Keycloak pods available as Endpoints to an AWS Global Accelerator instance. Each cluster endpoint is assigned a weight of 128 (half of the maximum weight 255) to ensure that accelerator traffic is routed equally to both availability-zones when both clusters are healthy.

11.3. PREREQUISITES

- ROSA based Multi-AZ Red Hat build of Keycloak deployment

11.4. PROCEDURE

1. Create Network Load Balancers
   Perform the following on each of the Red Hat build of Keycloak clusters:

   a. Login to the ROSA cluster

   b. Create a Kubernetes load balancer service

   Command:

```
cat <<EOF | oc apply -n $NAMESPACE -f - (1)
  apiVersion: v1
  kind: Service
  metadata:
    name: accelerator-loadbalancer
    annotations:
      service.beta.kubernetes.io/aws-load-balancer-additional-resource-tags:
accelerator=${ACCELERATOR_NAME},site=${CLUSTER_NAME},namespace=${NAME
SPACE} (2)
      service.beta.kubernetes.io/aws-load-balancer-type: "nlb"
      service.beta.kubernetes.io/aws-load-balancer-healthcheck-path: "/lb-check"
      service.beta.kubernetes.io/aws-load-balancer-healthcheck-protocol: "https"
      service.beta.kubernetes.io/aws-load-balancer-healthcheck-interval: "10" (3)
      service.beta.kubernetes.io/aws-load-balancer-healthcheck-healthy-threshold: "3" (4)
      service.beta.kubernetes.io/aws-load-balancer-healthcheck-unhealthy-threshold: "3"
(5)
  spec:
    ports:
    - name: https
      port: 443
      protocol: TCP
      targetPort: 8443
    selector:
      app: keycloak
      app.kubernetes.io/instance: keycloak
      app.kubernetes.io/managed-by: keycloak-operator
    sessionAffinity: None
    type: LoadBalancer
EOF
```

(1) **$NAMESPACE** should be replaced with the namespace of your Red Hat build of Keycloak deployment

(2) Add additional Tags to the resources created by AWS so that we can retrieve them later. **ACCELERATOR_NAME** should be the name of the Global Accelerator created in subsequent steps and **CLUSTER_NAME** should be the name of the current site.

(3) How frequently the healthcheck probe is executed in seconds

(4) How many healthchecks must pass for the NLB to be considered healthy

5 How many healthchecks must fail for the NLB to be considered unhealthy

c. Take note of the DNS hostname as this will be required later:

Command:

```
oc -n $NAMESPACE get svc accelerator-loadbalancer --template="{{range
.status.loadBalancer.ingress}}{{.hostname}}{{end}}"
```

Output:

```
abab80a363ce8479ea9c4349d116bce2-6b65e8b4272fa4b5.elb.eu-west-
1.amazonaws.com
```

2. Create a Global Accelerator instance

Command:

```
aws globalaccelerator create-accelerator \
  --name example-accelerator \ 1
  --ip-address-type DUAL_STACK \ 2
  --region us-west-2 3
```

1 The name of the accelerator to be created, update as required

2 Can be 'DUAL_STACK' or 'IPV4'

3 All **globalaccelerator** commands must use the region 'us-west-2'

Output:

```
{
    "Accelerator": {
        "AcceleratorArn": "arn:aws:globalaccelerator::606671647913:accelerator/e35a94dd-
391f-4e3e-9a3d-d5ad22a78c71", 1
        "Name": "example-accelerator",
        "IpAddressType": "DUAL_STACK",
        "Enabled": true,
        "IpSets": [
            {
                "IpFamily": "IPv4",
                "IpAddresses": [
                    "75.2.42.125",
                    "99.83.132.135"
                ],
                "IpAddressFamily": "IPv4"
            },
            {
                "IpFamily": "IPv6",
                "IpAddresses": [
                    "2600:9000:a400:4092:88f3:82e2:e5b2:e686",
                    "2600:9000:a516:b4ef:157e:4cbd:7b48:20f1"
```

```
            ],
            "IpAddressFamily": "IPv6"
        }
    ],
    "DnsName": "a099f799900e5b10d.awsglobalaccelerator.com", (2)
    "Status": "IN_PROGRESS",
    "CreatedTime": "2023-11-13T15:46:40+00:00",
    "LastModifiedTime": "2023-11-13T15:46:42+00:00",
    "DualStackDnsName": "ac86191ca5121e885.dualstack.awsglobalaccelerator.com" (3)
  }
}
```

(1) The ARN associated with the created Accelerator instance, this will be used in subsequent commands

(2) The DNS name which IPv4 Red Hat build of Keycloak clients should connect to

(3) The DNS name which IPv6 Red Hat build of Keycloak clients should connect to

3. Create a Listener for the accelerator

Command:

```
aws globalaccelerator create-listener \
  --accelerator-arn 'arn:aws:globalaccelerator::606671647913:accelerator/e35a94dd-391f-4e3e-9a3d-d5ad22a78c71' \
  --port-ranges '[{"FromPort":443,"ToPort":443}]' \
  --protocol TCP \
  --region us-west-2
```

Output:

```
{
  "Listener": {
    "ListenerArn": "arn:aws:globalaccelerator::606671647913:accelerator/e35a94dd-391f-4e3e-9a3d-d5ad22a78c71/listener/1f396d40",
    "PortRanges": [
      {
        "FromPort": 443,
        "ToPort": 443
      }
    ],
    "Protocol": "TCP",
    "ClientAffinity": "NONE"
  }
}
```

4. Create an Endpoint Group for the Listener

Command:

```
CLUSTER_1_ENDPOINT_ARN=$(aws elbv2 describe-load-balancers \
  --query "LoadBalancers[?DNSName=='abab80a363ce8479ea9c4349d116bce2-
```

```
6b65e8b4272fa4b5.elb.eu-west-1.amazonaws.com'].LoadBalancerArn" \ 1
   --region eu-west-1 \ 2
   --output text
)
CLUSTER_2_ENDPOINT_ARN=$(aws elbv2 describe-load-balancers \
   --query "LoadBalancers[?DNSName=='a1c76566e3c334e4ab7b762d9f8dcbcf-
985941f9c8d108d4.elb.eu-west-1.amazonaws.com'].LoadBalancerArn" \ 3
   --region eu-west-1 \ 4
   --output text
)
ENDPOINTS='[
  {
    "EndpointId": "'${CLUSTER_1_ENDPOINT_ARN}'",
    "Weight": 128,
    "ClientIPPreservationEnabled": false
  },
  {
    "EndpointId": "'${CLUSTER_2_ENDPOINT_ARN}'",
    "Weight": 128,
    "ClientIPPreservationEnabled": false
  }
]'
aws globalaccelerator create-endpoint-group \
  --listener-arn 'arn:aws:globalaccelerator::606671647913:accelerator/e35a94dd-391f-4e3e-
9a3d-d5ad22a78c71/listener/1f396d40' \ 5
  --traffic-dial-percentage 100 \
  --endpoint-configurations ${ENDPOINTS} \
  --endpoint-group-region eu-west-1 \ 6
  --region us-west-2
```

1 3 The DNS hostname of the Cluster's NLB

2 4 5 The ARN of the Listener created in the previous step

6 This should be the AWS region that hosts the clusters

Output:

```
{
    "EndpointGroup": {
        "EndpointGroupArn": "arn:aws:globalaccelerator::606671647913:accelerator/e35a94dd-
391f-4e3e-9a3d-d5ad22a78c71/listener/1f396d40/endpoint-group/2581af0dc700",
        "EndpointGroupRegion": "eu-west-1",
        "EndpointDescriptions": [
            {
                "EndpointId": "arn:aws:elasticloadbalancing:eu-west-
1:606671647913:loadbalancer/net/abab80a363ce8479ea9c4349d116bce2/6b65e8b4272fa4b5
",
                "Weight": 128,
                "HealthState": "HEALTHY",
                "ClientIPPreservationEnabled": false
            },
            {
                "EndpointId": "arn:aws:elasticloadbalancing:eu-west-
```

```
1:606671647913:loadbalancer/net/a1c76566e3c334e4ab7b762d9f8dcbcf/985941f9c8d108d4"
,
            "Weight": 128,
            "HealthState": "HEALTHY",
            "ClientIPPreservationEnabled": false
          }
      ],
      "TrafficDialPercentage": 100.0,
      "HealthCheckPort": 443,
      "HealthCheckProtocol": "TCP",
      "HealthCheckPath": "undefined",
      "HealthCheckIntervalSeconds": 30,
      "ThresholdCount": 3
    }
}
```

5. Optional: Configure your custom domain
   If you are using a custom domain, pointed your custom domain to the AWS Global Load Balancer by configuring an Alias or CNAME in your custom domain.

6. Create or update the Red Hat build of Keycloak Deployment
   Perform the following on each of the Red Hat build of Keycloak clusters:

   a. Login to the ROSA cluster

   b. Ensure the Keycloak CR has the following configuration

```
apiVersion: k8s.keycloak.org/v2alpha1
kind: Keycloak
metadata:
  name: keycloak
spec:
  hostname:
    hostname: $HOSTNAME (1)
  ingress:
    enabled: false (2)
```

(1) The hostname clients use to connect to Keycloak

(2) Disable the default ingress as all Red Hat build of Keycloak access should be via the provisioned NLB

To ensure that request forwarding works as expected, it is necessary for the Keycloak CR to specify the hostname through which clients will access the Red Hat build of Keycloak instances. This can either be the **DualStackDnsName** or **DnsName** hostname associated with the Global Accelerator. If you are using a custom domain, point your custom domain to the AWS Global Accelerator, and use your custom domain here.

11.5. VERIFY

To verify that the Global Accelerator is correctly configured to connect to the clusters, navigate to hostname configured above, and you should be presented with the Red Hat build of Keycloak admin console.

11.6. FURTHER READING

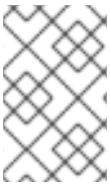
- Bringing a site online
- Taking a site offline

CHAPTER 12. DEPLOYING AN AWS LAMBDA TO DISABLE A NON-RESPONDING SITE

Deploy an AWS Lambda as part of the load-balancer building block in a multi-site deployment.

This chapter explains how to resolve split-brain scenarios between two sites in a multi-site deployment. It also disables replication if one site fails, so the other site can continue to serve requests.

This deployment is intended to be used with the setup described in the Concepts for multi-site deployments chapter. Use this deployment with the other building blocks outlined in the Building blocks multi-site deployments chapter.



NOTE

We provide these blueprints to show a minimal functionally complete example with a good baseline performance for regular installations. You would still need to adapt it to your environment and your organization's standards and security best practices.

12.1. ARCHITECTURE

In the event of a network communication failure between sites in a multi-site deployment, it is no longer possible for the two sites to continue to replicate the data between them. The Data Grid is configured with a **FAIL** failure policy, which ensures consistency over availability. Consequently, all user requests are served with an error message until the failure is resolved, either by restoring the network connection or by disabling cross-site replication.

In such scenarios, a quorum is commonly used to determine which sites are marked as online or offline. However, as multi-site deployments only consist of two sites, this is not possible. Instead, we leverage "fencing" to ensure that when one of the sites is unable to connect to the other site, only one site remains in the load balancer configuration, and hence only this site is able to serve subsequent users requests.

In addition to the load balancer configuration, the fencing procedure disables replication between the two Data Grid clusters to allow serving user requests from the site that remains in the load balancer configuration. As a result, the sites will be out-of-sync once the replication has been disabled.

To recover from the out-of-sync state, a manual re-sync is necessary as described in Synchronizing sites. This is why a site which is removed via fencing will not be re-added automatically when the network communication failure is resolved. The remove site should only be re-added once the two sites have been synchronized using the outlined procedure Bringing a site online .

In this chapter we describe how to implement fencing using a combination of Prometheus Alerts and AWS Lambda functions. A Prometheus Alert is triggered when split-brain is detected by the Data Grid server metrics, which results in the Prometheus AlertManager calling the AWS Lambda based webhook. The triggered Lambda function inspects the current Global Accelerator configuration and removes the site reported to be offline.

In a true split-brain scenario, where both sites are still up but network communication is down, it is possible that both sites will trigger the webhook simultaneously. We guard against this by ensuring that only a single Lambda instance can be executed at a given time. The logic in the AWS Lambda ensures that always one site entry remains in the load balancer configuration.

12.2. PREREQUISITES

- ROSA HCP based multi-site Keycloak deployment
- AWS CLI Installed
- AWS Global Accelerator load balancer
- **jq** tool installed

12.3. PROCEDURE

1. Enable Openshift user alert routing

   Command:

```
oc apply -f - << EOF
apiVersion: v1
kind: ConfigMap
metadata:
  name: user-workload-monitoring-config
  namespace: openshift-user-workload-monitoring
data:
  config.yaml: |
    alertmanager:
      enabled: true
      enableAlertmanagerConfig: true
EOF
oc -n openshift-user-workload-monitoring rollout status --watch
statefulset.apps/alertmanager-user-workload
```

2. Decide upon a username/password combination which will be used to authenticate the Lambda webhook and create an AWS Secret storing the password

   Command:

```
aws secretsmanager create-secret \
  --name webhook-password \ 1
  --secret-string changeme \ 2
  --region eu-west-1 3
```

   1 The name of the secret

   2 The password to be used for authentication

   3 The AWS region that hosts the secret

3. Create the Role used to execute the Lambda.

   Command:

```
FUNCTION_NAME= 1
ROLE_ARN=$(aws iam create-role \
  --role-name ${FUNCTION_NAME} \
  --assume-role-policy-document \
```

```
  '{
    "Version": "2012-10-17",
    "Statement": [
      {
        "Effect": "Allow",
        "Principal": {
          "Service": "lambda.amazonaws.com"
        },
        "Action": "sts:AssumeRole"
      }
    ]
  }' \
  --query 'Role.Arn' \
  --region eu-west-1 \ 2
  --output text
)
```

1 A name of your choice to associate with the Lambda and related resources

2 The AWS Region hosting your Kubernetes clusters

4. Create and attach the 'LambdaSecretManager' Policy so that the Lambda can access AWS Secrets

   Command:

```
POLICY_ARN=$(aws iam create-policy \
  --policy-name LambdaSecretManager \
  --policy-document \
  '{
     "Version": "2012-10-17",
     "Statement": [
        {
           "Effect": "Allow",
           "Action": [
              "secretsmanager:GetSecretValue"
           ],
           "Resource": "*"
        }
     ]
  }' \
  --query 'Policy.Arn' \
  --output text
)
aws iam attach-role-policy \
  --role-name ${FUNCTION_NAME} \
  --policy-arn ${POLICY_ARN}
```

5. Attach the **ElasticLoadBalancingReadOnly** policy so that the Lambda can query the provisioned Network Load Balancers

   Command:

```
aws iam attach-role-policy \
  --role-name ${FUNCTION_NAME} \
  --policy-arn arn:aws:iam::aws:policy/ElasticLoadBalancingReadOnly
```

6. Attach the **GlobalAcceleratorFullAccess** policy so that the Lambda can update the Global Accelerator EndpointGroup

   Command:

```
aws iam attach-role-policy \
  --role-name ${FUNCTION_NAME} \
  --policy-arn arn:aws:iam::aws:policy/GlobalAcceleratorFullAccess
```

7. Create a Lambda ZIP file containing the required fencing logic

   Command:

```
LAMBDA_ZIP=/tmp/lambda.zip
cat << EOF > /tmp/lambda.py

from urllib.error import HTTPError

import boto3
import jmespath
import json
import os
import urllib3

from base64 import b64decode
from urllib.parse import unquote

# Prevent unverified HTTPS connection warning
urllib3.disable_warnings(urllib3.exceptions.InsecureRequestWarning)


class MissingEnvironmentVariable(Exception):
    pass


class MissingSiteUrl(Exception):
    pass


def env(name):
    if name in os.environ:
        return os.environ[name]
    raise MissingEnvironmentVariable(f"Environment Variable '{name}' must be set")


def handle_site_offline(labels):
    a_client = boto3.client('globalaccelerator', region_name='us-west-2')

    acceleratorDNS = labels['accelerator']
    accelerator = jmespath.search(f"Accelerators[?(DnsName=='{acceleratorDNS}'||
DualStackDnsName=='{acceleratorDNS}')]", a_client.list_accelerators())
```

```
    if not accelerator:
        print(f"Ignoring SiteOffline alert as accelerator with DnsName '{acceleratorDNS}' not
found")
        return

    accelerator_arn = accelerator[0]['AcceleratorArn']
    listener_arn = a_client.list_listeners(AcceleratorArn=accelerator_arn)['Listeners'][0]
['ListenerArn']

    endpoint_group = a_client.list_endpoint_groups(ListenerArn=listener_arn)
['EndpointGroups'][0]
    endpoints = endpoint_group['EndpointDescriptions']

    # Only update accelerator endpoints if two entries exist
    if len(endpoints) > 1:
        # If the reporter endpoint is not healthy then do nothing for now
        # A Lambda will eventually be triggered by the other offline site for this reporter
        reporter = labels['reporter']
        reporter_endpoint = [e for e in endpoints if endpoint_belongs_to_site(e, reporter)][0]
        if reporter_endpoint['HealthState'] == 'UNHEALTHY':
            print(f"Ignoring SiteOffline alert as reporter '{reporter}' endpoint is marked
UNHEALTHY")
            return

        offline_site = labels['site']
        endpoints = [e for e in endpoints if not endpoint_belongs_to_site(e, offline_site)]
        del reporter_endpoint['HealthState']
        a_client.update_endpoint_group(
            EndpointGroupArn=endpoint_group['EndpointGroupArn'],
            EndpointConfigurations=endpoints
        )
        print(f"Removed site={offline_site} from Accelerator EndpointGroup")

        take_infinispan_site_offline(reporter, offline_site)
        print(f"Backup site={offline_site} caches taken offline")
    else:
        print("Ignoring SiteOffline alert only one Endpoint defined in the EndpointGroup")


def endpoint_belongs_to_site(endpoint, site):
    lb_arn = endpoint['EndpointId']
    region = lb_arn.split(':')[3]
    client = boto3.client('elbv2', region_name=region)
    tags = client.describe_tags(ResourceArns=[lb_arn])['TagDescriptions'][0]['Tags']
    for tag in tags:
        if tag['Key'] == 'site':
            return tag['Value'] == site
    return false


def take_infinispan_site_offline(reporter, offlinesite):
    endpoints = json.loads(INFINISPAN_SITE_ENDPOINTS)
    if reporter not in endpoints:
        raise MissingSiteUrl(f"Missing URL for site '{reporter}' in
'INFINISPAN_SITE_ENDPOINTS' json")
```

```
    endpoint = endpoints[reporter]
    password = get_secret(INFINISPAN_USER_SECRET)
    url = f"https://{endpoint}/rest/v2/container/x-site/backups/{offlinesite}?action=take-offline"
    http = urllib3.PoolManager(cert_reqs='CERT_NONE')
    headers = urllib3.make_headers(basic_auth=f"{INFINISPAN_USER}:{password}")
    try:
        rsp = http.request("POST", url, headers=headers)
        if rsp.status >= 400:
            raise HTTPError(f"Unexpected response status '%d' when taking site offline",
rsp.status)
        rsp.release_conn()
    except HTTPError as e:
        print(f"HTTP error encountered: {e}")


def get_secret(secret_name):
    session = boto3.session.Session()
    client = session.client(
        service_name='secretsmanager',
        region_name=SECRETS_REGION
    )
    return client.get_secret_value(SecretId=secret_name)['SecretString']


def decode_basic_auth_header(encoded_str):
    split = encoded_str.strip().split(' ')
    if len(split) == 2:
        if split[0].strip().lower() == 'basic':
            try:
                username, password = b64decode(split[1]).decode().split(':', 1)
            except:
                raise DecodeError
        else:
            raise DecodeError
    else:
        raise DecodeError

    return unquote(username), unquote(password)


def handler(event, context):
    print(json.dumps(event))

    authorization = event['headers'].get('authorization')
    if authorization is None:
        print("'Authorization' header missing from request")
        return {
            "statusCode": 401
        }

    expectedPass = get_secret(WEBHOOK_USER_SECRET)
    username, password = decode_basic_auth_header(authorization)
    if username != WEBHOOK_USER and password != expectedPass:
        print('Invalid username/password combination')
        return {
            "statusCode": 403
```

```
        }

    body = event.get('body')
    if body is None:
        raise Exception('Empty request body')

    body = json.loads(body)
    print(json.dumps(body))

    if body['status'] != 'firing':
        print("Ignoring alert as status is not 'firing', status was: '%s'" % body['status'])
        return {
            "statusCode": 204
        }

    for alert in body['alerts']:
        labels = alert['labels']
        if labels['alertname'] == 'SiteOffline':
            handle_site_offline(labels)

    return {
        "statusCode": 204
    }


INFINISPAN_USER = env('INFINISPAN_USER')
INFINISPAN_USER_SECRET = env('INFINISPAN_USER_SECRET')
INFINISPAN_SITE_ENDPOINTS = env('INFINISPAN_SITE_ENDPOINTS')
SECRETS_REGION = env('SECRETS_REGION')
WEBHOOK_USER = env('WEBHOOK_USER')
WEBHOOK_USER_SECRET = env('WEBHOOK_USER_SECRET')

EOF
zip -FS --junk-paths ${LAMBDA_ZIP} /tmp/lambda.py
```

8. Create the Lambda function.

Command:

```
aws lambda create-function \
  --function-name ${FUNCTION_NAME} \
  --zip-file fileb://${LAMBDA_ZIP} \
  --handler lambda.handler \
  --runtime python3.12 \
  --role ${ROLE_ARN} \
  --region eu-west-1 (1)
```

(1) The AWS Region hosting your Kubernetes clusters

9. Expose a Function URL so the Lambda can be triggered as webhook

Command:

```
aws lambda create-function-url-config \
```

```
--function-name ${FUNCTION_NAME} \
--auth-type NONE \
--region eu-west-1 (1)
```

(1) The AWS Region hosting your Kubernetes clusters

10. Allow public invocations of the Function URL

Command:

```
aws lambda add-permission \
  --action "lambda:InvokeFunctionUrl" \
  --function-name ${FUNCTION_NAME} \
  --principal "*" \
  --statement-id FunctionURLAllowPublicAccess \
  --function-url-auth-type NONE \
  --region eu-west-1 (1)
```

(1) The AWS Region hosting your Kubernetes clusters

11. Configure the Lambda's Environment variables:

    a. In each Kubernetes cluster, retrieve the exposed Data Grid URL endpoint:

```
oc -n ${NAMESPACE} get route infinispan-external -o jsonpath='{.status.ingress[].host}' (1)
```

(1) Replace **${NAMESPACE}** with the namespace containing your Data Grid server

    b. Upload the desired Environment variables

```
ACCELERATOR_NAME= (1)
LAMBDA_REGION= (2)
CLUSTER_1_NAME= (3)
CLUSTER_1_ISPN_ENDPOINT= (4)
CLUSTER_2_NAME= (5)
CLUSTER_2_ISPN_ENDPOINT= (6)
INFINISPAN_USER= (7)
INFINISPAN_USER_SECRET= (8)
WEBHOOK_USER= (9)
WEBHOOK_USER_SECRET= (10)

INFINISPAN_SITE_ENDPOINTS=$(echo "
{\"${CLUSTER_NAME_1}\":\"${CLUSTER_1_ISPN_ENDPOINT}\",\"${CLUSTER_2_NA
ME}\":\"${CLUSTER_2_ISPN_ENDPOINT\"}" | jq tostring)
aws lambda update-function-configuration \
    --function-name ${ACCELERATOR_NAME} \
    --region ${LAMBDA_REGION} \
    --environment "{
      \"Variables\": {
        \"INFINISPAN_USER\" : \"${INFINISPAN_USER}\",
```

```
    \"INFINISPAN_USER_SECRET\" : \"${INFINISPAN_USER_SECRET}\",
    \"INFINISPAN_SITE_ENDPOINTS\" : ${INFINISPAN_SITE_ENDPOINTS},
    \"WEBHOOK_USER\" : \"${WEBHOOK_USER}\",
    \"WEBHOOK_USER_SECRET\" : \"${WEBHOOK_USER_SECERT}\",
    \"SECRETS_REGION\" : \"eu-central-1\"
  }
}"
```

1. The name of the AWS Global Accelerator used by your deployment
2. The AWS Region hosting your Kubernetes cluster and Lambda function
3. The name of one of your Data Grid sites as defined in Deploying Data Grid for HA with the Data Grid Operator
4. The Data Grid endpoint URL associated with the CLUSER_1_NAME site
5. The name of the second Data Grid site
6. The Data Grid endpoint URL associated with the CLUSER_2_NAME site
7. The username of a Data Grid user which has sufficient privileges to perform REST requests on the server
8. The name of the AWS secret containing the password associated with the Data Grid user
9. The username used to authenticate requests to the Lambda Function
10. The name of the AWS secret containing the password used to authenticate requests to the Lambda function

12. Retrieve the Lambda Function URL

Command:

```
aws lambda get-function-url-config \
  --function-name ${FUNCTION_NAME} \
  --query "FunctionUrl" \
  --region eu-west-1 \ 1
  --output text
```

1. The AWS region where the Lambda was created

Output:

```
https://tjqr2vgc664b6noj6vugprakoq0ausj.lambda-url.eu-west-1.on.aws
```

13. In each Kubernetes cluster, configure a Prometheus Alert routing to trigger the Lambda on split-brain

Command:

```
NAMESPACE= # The namespace containing your deployments
```

```
oc apply -n ${NAMESPACE} -f - << EOF
apiVersion: v1
kind: Secret
type: kubernetes.io/basic-auth
metadata:
  name: webhook-credentials
stringData:
  username: 'keycloak' (1)
  password: 'changme' (2)
---
apiVersion: monitoring.coreos.com/v1beta1
kind: AlertmanagerConfig
metadata:
  name: example-routing
spec:
  route:
    receiver: default
    groupBy:
      - accelerator
    groupInterval: 90s
    groupWait: 60s
    matchers:
      - matchType: =
        name: alertname
        value: SiteOffline
  receivers:
    - name: default
      webhookConfigs:
        - url: 'https://tjqr2vgc664b6noj6vugprakoq0oausj.lambda-url.eu-west-1.on.aws/' (3)
          httpConfig:
            basicAuth:
              username:
                key: username
                name: webhook-credentials
              password:
                key: password
                name: webhook-credentials
            tlsConfig:
              insecureSkipVerify: true
---
apiVersion: monitoring.coreos.com/v1
kind: PrometheusRule
metadata:
  name: xsite-status
spec:
  groups:
    - name: xsite-status
      rules:
        - alert: SiteOffline
          expr: 'min by (namespace, site)
(vendor_jgroups_site_view_status{namespace="default",site="site-b"}) == 0' (4)
          labels:
            severity: critical
            reporter: site-a (5)
            accelerator: a3da6a6cbd4e27b02.awsglobalaccelerator.com (6)
```

1. The username required to authenticate Lambda requests
2. The password required to authenticate Lambda requests
3. The Lambda Function URL
4. The namespace value should be the namespace hosting the Infinispan CR and the site should be the remote site defined by **spec.service.sites.locations[0].name** in your Infinispan CR
5. The name of your local site defined by **spec.service.sites.local.name** in your Infinispan CR
6. The DNS of your Global Accelerator

12.4. VERIFY

To test that the Prometheus alert triggers the webhook as expected, perform the following steps to simulate a split-brain:

1. In each of your clusters execute the following:

   Command:

```
oc -n openshift-operators scale --replicas=0 deployment/infinispan-operator-controller-manager (1)
oc -n openshift-operators rollout status -w deployment/infinispan-operator-controller-manager
oc -n ${NAMESPACE} scale --replicas=0 deployment/infinispan-router (2)
oc -n ${NAMESPACE} rollout status -w deployment/infinispan-router
```

   1. Scale down the Data Grid Operator so that the next step does not result in the deployment being recreated by the operator
   2. Scale down the Gossip Router deployment.Replace **${NAMESPACE}** with the namespace containing your Data Grid server

2. Verify the **SiteOffline** event has been fired on a cluster by inspecting the **Observe → Alerting** menu in the Openshift console
3. Inspect the Global Accelerator EndpointGroup in the AWS console and there should only be a single endpoint present
4. Scale up the Data Grid Operator and Gossip Router to re-establish a connection between sites:

   Command:

```
oc -n openshift-operators scale --replicas=1 deployment/infinispan-operator-controller-manager
oc -n openshift-operators rollout status -w deployment/infinispan-operator-controller-manager
oc -n ${NAMESPACE} scale --replicas=1 deployment/infinispan-router (1)
oc -n ${NAMESPACE} rollout status -w deployment/infinispan-router
```

1 Replace **${NAMESPACE}** with the namespace containing your Data Grid server

5. Inspect the **vendor_jgroups_site_view_status** metric in each site. A value of **1** indicates that the site is reachable.

6. Update the Accelerator EndpointGroup to contain both Endpoints. See the Bringing a site online chapter for details.

12.5. FURTHER READING

- Bringing a site online
- Taking a site offline

CHAPTER 13. TAKING A SITE OFFLINE

Take a site offline so that it no longer processes client requests.

13.1. WHEN TO USE THIS PROCEDURE

During the deployment lifecycle it might be required that one of the sites is temporarily taken offline for maintenance or to allow for software upgrades. To ensure that no user requests are routed to the site requiring maintenance, it is necessary for the site to be removed from your load balancer configuration.

13.2. PROCEDURE

Follow these steps to remove a site from the load balancer so that no traffic can be routed to it.

13.2.1. Global Accelerator

1. Determine the ARN of the Network Load Balancer (NLB) associated with the site to be kept online

   Command:

   ```
   NAMESPACE= 1
   REGION= 2
   HOSTNAME=$(oc -n $NAMESPACE get svc accelerator-loadbalancer --template="{{range
   .status.loadBalancer.ingress}}{{.hostname}}{{end}}")
   aws elbv2 describe-load-balancers \
     --query "LoadBalancers[?DNSName=='${HOSTNAME}'].LoadBalancerArn" \
     --region ${REGION} \
     --output text
   ```

   1 The Kubernetes namespace containing the Keycloak deployment

   2 The AWS Region hosting the Kubernetes cluster

   Output:

   ```
   arn:aws:elasticloadbalancing:eu-west-
   1:606671647913:loadbalancer/net/a49e56e51e16843b9a3bc686327c907b/9b786f80ed4eba3d
   ```

2. Update the Accelerator EndpointGroup to only include a single site

   a. List the current endpoints in the Global Accelerator's EndpointGroup

      Command:

      ```
      ACCELERATOR_NAME= 1
      ACCELERATOR_ARN=$(aws globalaccelerator list-accelerators \
        --query "Accelerators[?Name=='${ACCELERATOR_NAME}'].AcceleratorArn" \
        --region us-west-2 \ 2
        --output text
      )
      LISTENER_ARN=$(aws globalaccelerator list-listeners \
      ```

```
  --accelerator-arn ${ACCELERATOR_ARN} \
  --query "Listeners[*].ListenerArn" \
  --region us-west-2 \
  --output text
)
aws globalaccelerator list-endpoint-groups \
  --listener-arn ${LISTENER_ARN} \
  --region us-west-2
```

1. The name of the Accelerator to be updated
2. The region must always be set to us-west-2 when querying AWS Global Accelerators

Output:


```
{
    "EndpointGroups": [
        {
            "EndpointGroupArn":
"arn:aws:globalaccelerator::606671647913:accelerator/d280fc09-3057-4ab6-9330-
6cbf1f450748/listener/8769072f/endpoint-group/a30b64ec1700",
            "EndpointGroupRegion": "eu-west-1",
            "EndpointDescriptions": [
                {
                    "EndpointId": "arn:aws:elasticloadbalancing:eu-west-
1:606671647913:loadbalancer/net/a49e56e51e16843b9a3bc686327c907b/9b786f80ed4e
ba3d",
                    "Weight": 128,
                    "HealthState": "HEALTHY",
                    "ClientIPPreservationEnabled": false
                },
                {
                    "EndpointId": "arn:aws:elasticloadbalancing:eu-west-
1:606671647913:loadbalancer/net/a3c75f239541c4a6e9c48cf8d48d602f/5ba333e87019ccf
0",
                    "Weight": 128,
                    "HealthState": "HEALTHY",
                    "ClientIPPreservationEnabled": false
                }
            ],
            "TrafficDialPercentage": 100.0,
            "HealthCheckPort": 443,
            "HealthCheckProtocol": "TCP",
            "HealthCheckIntervalSeconds": 30,
            "ThresholdCount": 3
        }
    ]
}
```


b. Update the EndpointGroup to only include the NLB retrieved in step 1.

Command:

```
aws globalaccelerator update-endpoint-group \
```

```
  --endpoint-group-arn arn:aws:globalaccelerator::606671647913:accelerator/d280fc09-
3057-4ab6-9330-6cbf1f450748/listener/8769072f/endpoint-group/a30b64ec1700 \
  --region us-west-2 \
  --endpoint-configurations '
  [
    {
        "EndpointId": "arn:aws:elasticloadbalancing:eu-west-
1:606671647913:loadbalancer/net/a49e56e51e16843b9a3bc686327c907b/9b786f80ed4e
ba3d",
        "Weight": 128,
        "ClientIPPreservationEnabled": false
    }
  ]
'
```

CHAPTER 14. BRINGING A SITE ONLINE

Bring a site online so that it can process client requests.

14.1. WHEN TO USE THIS PROCEDURE

This procedure describes how to re-add a Keycloak site to the Global Accelerator, after it has previously been taken offline, so that it can once again service client requests.

14.2. PROCEDURE

Follow these steps to re-add a Keycloak site to the AWS Global Accelerator so that it can handle client requests.

14.2.1. Global Accelerator

1. Determine the ARN of the Network Load Balancer (NLB) associated with the site to be brought online

   Command:

   ```
   NAMESPACE= 1
   REGION= 2
   HOSTNAME=$(oc -n $NAMESPACE get svc accelerator-loadbalancer --template="{{range
   .status.loadBalancer.ingress}}{{.hostname}}{{end}}")
   aws elbv2 describe-load-balancers \
     --query "LoadBalancers[?DNSName=='${HOSTNAME}'].LoadBalancerArn" \
     --region ${REGION} \
     --output text
   ```

   1 The Kubernetes namespace containing the Keycloak deployment

   2 The AWS Region hosting the Kubernetes cluster

   Output:

   ```
   arn:aws:elasticloadbalancing:eu-west-
   1:606671647913:loadbalancer/net/a49e56e51e16843b9a3bc686327c907b/9b786f80ed4eba3d
   ```

2. Update the Accelerator EndpointGroup to include both sites

   a. List the current endpoints in the Global Accelerator's EndpointGroup

      Command:

      ```
      ACCELERATOR_NAME= 1
      ACCELERATOR_ARN=$(aws globalaccelerator list-accelerators \
        --query "Accelerators[?Name=='${ACCELERATOR_NAME}'].AcceleratorArn" \
        --region us-west-2 \ 2
        --output text
      )
      LISTENER_ARN=$(aws globalaccelerator list-listeners \
      ```

```
  --accelerator-arn ${ACCELERATOR_ARN} \
  --query "Listeners[*].ListenerArn" \
  --region us-west-2 \
  --output text
)
aws globalaccelerator list-endpoint-groups \
  --listener-arn ${LISTENER_ARN} \
  --region us-west-2
```

1. The name of the Accelerator to be updated
2. The region must always be set to us-west-2 when querying AWS Global Accelerators

Output:

```
{
    "EndpointGroups": [
        {
            "EndpointGroupArn":
"arn:aws:globalaccelerator::606671647913:accelerator/d280fc09-3057-4ab6-9330-
6cbf1f450748/listener/8769072f/endpoint-group/a30b64ec1700",
            "EndpointGroupRegion": "eu-west-1",
            "EndpointDescriptions": [
                {
                    "EndpointId": "arn:aws:elasticloadbalancing:eu-west-
1:606671647913:loadbalancer/net/a3c75f239541c4a6e9c48cf8d48d602f/5ba333e87019ccf
0",
                    "Weight": 128,
                    "HealthState": "HEALTHY",
                    "ClientIPPreservationEnabled": false
                }
            ],
            "TrafficDialPercentage": 100.0,
            "HealthCheckPort": 443,
            "HealthCheckProtocol": "TCP",
            "HealthCheckIntervalSeconds": 30,
            "ThresholdCount": 3
        }
    ]
}
```

b. Update the EndpointGroup to include the existing Endpoint and the NLB retrieved in step 1.

Command:

```
aws globalaccelerator update-endpoint-group \
  --endpoint-group-arn arn:aws:globalaccelerator::606671647913:accelerator/d280fc09-
3057-4ab6-9330-6cbf1f450748/listener/8769072f/endpoint-group/a30b64ec1700 \
  --region us-west-2 \
  --endpoint-configurations '
  [
    {
      "EndpointId": "arn:aws:elasticloadbalancing:eu-west-
1:606671647913:loadbalancer/net/a3c75f239541c4a6e9c48cf8d48d602f/5ba333e87019ccf
```

```
0",
        "Weight": 128,
        "ClientIPPreservationEnabled": false
    },
    {
        "EndpointId": "arn:aws:elasticloadbalancing:eu-west-
1:606671647913:loadbalancer/net/a49e56e51e16843b9a3bc686327c907b/9b786f80ed4e
ba3d",
        "Weight": 128,
        "ClientIPPreservationEnabled": false
    }
  ]
'
```

CHAPTER 15. SYNCHRONIZING SITES

Synchronize an offline site with an online site.

15.1. WHEN TO USE THIS PROCEDURE

Use this when the state of Data Grid clusters of two sites become disconnected and the contents of the caches are out-of-sync. Perform this for example after a split-brain or when one site has been taken offline for maintenance.

At the end of the procedure, the data on the secondary site have been discarded and replaced by the data of the active site. All caches in the offline site are cleared to prevent invalid cache contents.

15.2. PROCEDURES

15.2.1. Data Grid Cluster

For the context of this chapter, **site-a** is the currently active site and **site-b** is an offline site that is not part of the AWS Global Accelerator EndpointGroup and is therefore not receiving user requests.

WARNING

Transferring state may impact Data Grid cluster performance by increasing the response time and/or resources usage.

The first procedure is to delete the stale data from the offline site.

1. Login into the offline site.
2. Shutdown Red Hat build of Keycloak. This will clear all Red Hat build of Keycloak caches and prevents the Red Hat build of Keycloak state from being out-of-sync with Data Grid.
   When deploying Red Hat build of Keycloak using the Red Hat build of Keycloak Operator, change the number of Red Hat build of Keycloak instances in the Red Hat build of Keycloak Custom Resource to 0.
3. Connect into Data Grid Cluster using the Data Grid CLI tool:

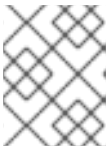
   Command:

   ```
   oc -n keycloak exec -it pods/infinispan-0 -- ./bin/cli.sh --trustall --connect
   https://127.0.0.1:11222
   ```

   It asks for the username and password for the Data Grid cluster. Those credentials are the one set in the Deploying Data Grid for HA with the Data Grid Operator chapter in the configuring credentials section.

   Output:

```
Username: developer
Password:
[infinispan-0-29897@ISPN//containers/default]>
```



NOTE

The pod name depends on the cluster name defined in the Data Grid CR. The connection can be done with any pod in the Data Grid cluster.

4. Disable the replication from offline site to the active site by running the following command. It prevents the clear request to reach the active site and delete all the correct cached data.

Command:

```
site take-offline --all-caches --site=site-a
```

Output:

```
{
  "authenticationSessions" : "ok",
  "work" : "ok",
  "loginFailures" : "ok",
  "actionTokens" : "ok"
}
```

5. Check the replication status is **offline**.

Command:

```
site status --all-caches --site=site-a
```

Output:

```
{
  "status" : "offline"
}
```

If the status is not **offline**, repeat the previous step.

WARNING

Make sure the replication is **offline** otherwise the clear data will clear both sites.

6. Clear all the cached data in offline site using the following commands:

Command:

```
clearcache actionTokens
clearcache authenticationSessions
clearcache loginFailures
clearcache work
```

These commands do not print any output.

7. Re-enable the cross-site replication from offline site to the active site.

Command:

```
site bring-online --all-caches --site=site-a
```

Output:

```
{
  "authenticationSessions" : "ok",
  "work" : "ok",
  "loginFailures" : "ok",
  "actionTokens" : "ok"
}
```

8. Check the replication status is **online**.

Command:

```
site status --all-caches --site=site-a
```

Output:

```
{
  "status" : "online"
}
```

Now we are ready to transfer the state from the active site to the offline site.

1. Login into your Active site
2. Connect into Data Grid Cluster using the Data Grid CLI tool:

Command:

```
oc -n keycloak exec -it pods/infinispan-0 -- ./bin/cli.sh --trustall --connect
https://127.0.0.1:11222
```

It asks for the username and password for the Data Grid cluster. Those credentials are the one set in the Deploying Data Grid for HA with the Data Grid Operator chapter in the configuring credentials section.

Output:

```
Username: developer
Password:
```

```
[infinispan-0-29897@ISPN//containers/default]>
```

NOTE

The pod name depends on the cluster name defined in the Data Grid CR. The connection can be done with any pod in the Data Grid cluster.

3. Trigger the state transfer from the active site to the offline site.

   Command:

   ```
   site push-site-state --all-caches --site=site-b
   ```

   Output:

   ```
   {
     "authenticationSessions" : "ok",
     "work" : "ok",
     "loginFailures" : "ok",
     "actionTokens" : "ok"
   }
   ```

4. Check the replication status is **online** for all caches.

   Command:

   ```
   site status --all-caches --site=site-b
   ```

   Output:

   ```
   {
     "status" : "online"
   }
   ```

5. Wait for the state transfer to complete by checking the output of **push-site-status** command for all caches.

   Command:

   ```
   site push-site-status --cache=actionTokens
   site push-site-status --cache=authenticationSessions
   site push-site-status --cache=loginFailures
   site push-site-status --cache=work
   ```

   Output:

   ```
   {
     "site-b" : "OK"
   }
   {
     "site-b" : "OK"
   }
   ```

```
{
  "site-b" : "OK"
}
{
  "site-b" : "OK"
}
```

Check the table in this section for the Cross-Site Documentation for the possible status values.

If an error is reported, repeat the state transfer for that specific cache.

Command:

```
site push-site-state --cache=<cache-name> --site=site-b
```

6. Clear/reset the state transfer status with the following command

Command:

```
site clear-push-site-status --cache=actionTokens
site clear-push-site-status --cache=authenticationSessions
site clear-push-site-status --cache=loginFailures
site clear-push-site-status --cache=work
```

Output:

```
"ok"
"ok"
"ok"
"ok"
```

Now the state is available in the offline site, Red Hat build of Keycloak can be started again:

1. Login into your secondary site.
2. Startup Red Hat build of Keycloak.
   When deploying Red Hat build of Keycloak using the Red Hat build of Keycloak Operator, change the number of Red Hat build of Keycloak instances in the Red Hat build of Keycloak Custom Resource to the original value.

15.2.2. AWS Aurora Database

No action required.

15.2.3. AWS Global Accelerator

Once the two sites have been synchronized, it is safe to add the previously offline site back to the Global Accelerator EndpointGroup following the steps in the Bringing a site online chapter.

15.3. FURTHER READING

See Concepts to automate Data Grid CLI commands .

CHAPTER 16. HEALTH CHECKS FOR MULTI-SITE DEPLOYMENTS

Validate the health of a multi-site deployment.

When running the Multi-site deployments in a Kubernetes environment, you should automate checks to see if everything is up and running as expected.

This page provides an overview of URLs, Kubernetes resources, and Healthcheck endpoints available to verify a multi-site setup of Red Hat build of Keycloak.

16.1. OVERVIEW

A proactive monitoring strategy aims to detect and alert about issues before they impact users. This strategy is the key for a highly resilient and highly available Red Hat build of Keycloak application.

Health checks across various architectural components (such as application health, load balancing, caching, and overall system status) are critical for:

Ensuring high availability

Verifying that all sites and the load balancer are operational is a key to ensure that a system can handle requests even if one site goes down.

Maintaining performance

Checking the health and distribution of the Data Grid cache ensures that Red Hat build of Keycloak can maintain optimal performance by efficiently handling sessions and other temporary data.

Operational resilience

By continuously monitoring the health of both Red Hat build of Keycloak and its dependencies within the Kubernetes environment, the system can quickly identify and possibly auto-remediate issues, reducing downtime.

16.2. PREREQUISITES

1. Kubectl CLI is installed and configured.
2. Install jq if it is not already installed on your operating system.

16.3. SPECIFIC HEALTH CHECKS

16.3.1. Red Hat build of Keycloak load balancer and sites

Verifies the health of the Red Hat build of Keycloak application through its load balancer and both primary and backup sites. This ensures that Red Hat build of Keycloak is accessible and that the load balancing mechanism is functioning correctly across different geographical or network locations.

This command returns the health status of the Red Hat build of Keycloak application's connection to its configured database, thus confirming the reliability of database connections. This command is available only on the management port and not from the external URL. In a Kubernetes setup, the sub-status **health/ready** is checked periodically to make the Pod as ready.

```
curl -s https://keycloak:managementport/health
```

This command verifies the **lb-check** endpoint of the load balancer and ensures the Red Hat build of Keycloak application cluster is up and running.

```
curl -s https://keycloak-load-balancer-url/lb-check
```

These commands will return the running status of the Site A and Site B of the Red Hat build of Keycloak in a multi-site setup.

```
curl -s https://keycloak_site_a_url/lb-check
curl -s https://keycloak_site_b_url/lb-check
```

16.3.2. Data Grid Cache health

Check the health of the default cache manager and individual caches in an external Data Grid cluster. This check is vital for Red Hat build of Keycloak performance and reliability, as Data Grid is often used for distributed caching and session clustering in Red Hat build of Keycloak deployments.

This command returns the overall health of the Data Grid cache manager, which is useful as the Admin user does not need to provide user credentials to get the health status.

```
curl -s https://infinispan_rest_url/rest/v2/cache-managers/default/health/status
```

In contrast to the preceding health checks, the following health checks require the Admin user to provide the Data Grid user credentials as part of the request to peek into the overall health of the external Data Grid cluster caches.

```
curl -u <infinispan_user>:<infinispan_pwd> -s https://infinispan_rest_url/rest/v2/cache-managers/default/health \
 | jq 'if .cluster_health.health_status == "HEALTHY" and (all(.cache_health[].status; . == "HEALTHY")) then "HEALTHY" else "UNHEALTHY" end'
```

The **jq** filter is a convenience to compute the overall health based on the individual cache health. You can also choose to run the above command without the **jq** filter to see the full details.

16.3.3. Data Grid Cluster distribution

Assesses the distribution health of the Data Grid cluster, ensuring that the cluster's nodes are correctly distributing data. This step is essential for the scalability and fault tolerance of the caching layer.

You can modify the **expectedCount 3** argument to match the total nodes in the cluster and validate if they are healthy or not.

```
curl <infinispan_user>:<infinispan_pwd> -s https://infinispan_rest_url/rest/v2/cluster\?action\=distribution \
 | jq --argjson expectedCount 3 'if map(select(.node_addresses | length > 0)) | length == $expectedCount then "HEALTHY" else "UNHEALTHY" end'
```

16.3.4. Overall, Data Grid system health

Uses the **oc** CLI tool to query the health status of Data Grid clusters and the Red Hat build of Keycloak service in the specified namespace. This comprehensive check ensures that all components of the Red Hat build of Keycloak deployment are operational and correctly configured within the Kubernetes

environment.

```
oc get infinispan -n <NAMESPACE> -o json  \
| jq '.items[].status.conditions' \
| jq 'map({(.type): .status})' \
| jq 'reduce .[] as $item ([]; . + [keys[] | select($item[.] != "True")]) | if length == 0 then "HEALTHY" else "UNHEALTHY: " + (join(", ")) end'
```

16.3.5. Red Hat build of Keycloak readiness in Kubernetes

Specifically, checks for the readiness and rolling update conditions of Red Hat build of Keycloak deployments in Kubernetes, ensuring that the Red Hat build of Keycloak instances are fully operational and not undergoing updates that could impact availability.

```
oc wait --for=condition=Ready --timeout=10s keycloaks.k8s.keycloak.org/keycloak -n <NAMESPACE>
oc wait --for=condition=RollingUpdate=False --timeout=10s keycloaks.k8s.keycloak.org/keycloak -n <NAMESPACE>
```