Red Hat build of Keycloak 24.0

Operator Guide

Last Updated: 2024-09-19

Legal Notice

Abstract

This guide consists of information for administrators to configure and use the Red Hat build of Keycloak 24.0 Operator.

Table of Contents

CHAPTER 1. RED HAT BUILD OF KEYCLOAK OPERATOR INSTALLATION

Use this procedure to install the Red Hat build of Keycloak Operator in an OpenShift cluster.

1. Open the OpenShift Container Platform web console.
2. In the left column, click **Home**, **Operators**, **OperatorHub**.
3. Search for "Keycloak" on the search input box.
4. Select the Operator from the list of results.
5. Follow the instructions on the screen.

For general instructions on installing Operators by using either the CLI or web console, see Installing Operators in your namespace. In the default Catalog, the Operator is named **rhbk-operator**. Make sure to use the channel corresponding with your desired Red Hat build of Keycloak version.

CHAPTER 2. BASIC RED HAT BUILD OF KEYCLOAK DEPLOYMENT

2.1. PERFORMING A BASIC RED HAT BUILD OF KEYCLOAK DEPLOYMENT

This chapter describes how to perform a basic Red Hat build of Keycloak Deployment on OpenShift using the Operator.

2.1.1. Preparing for deployment

Once the Red Hat build of Keycloak Operator is installed and running in the cluster namespace, you can set up the other deployment prerequisites.

- Database
- Hostname
- TLS Certificate and associated keys

2.1.1.1. Database

A database should be available and accessible from the cluster namespace where Red Hat build of Keycloak is installed. For a list of supported databases, see Configuring the database. The Red Hat build of Keycloak Operator does not manage the database and you need to provision it yourself. Consider verifying your cloud provider offering or using a database operator.

For development purposes, you can use an ephemeral PostgreSQL pod installation. To provision it, follow the approach below:

Create YAML file **example-postgres.yaml**:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: postgresql-db
spec:
  serviceName: postgresql-db-service
  selector:
    matchLabels:
      app: postgresql-db
  replicas: 1
  template:
    metadata:
      labels:
        app: postgresql-db
    spec:
      containers:
        - name: postgresql-db
          image: postgres:15
          volumeMounts:
            - mountPath: /data
              name: cache-volume
```

```
        env:
          - name: POSTGRES_USER
            value: testuser
          - name: POSTGRES_PASSWORD
            value: testpassword
          - name: PGDATA
            value: /data/pgdata
          - name: POSTGRES_DB
            value: keycloak
      volumes:
        - name: cache-volume
          emptyDir: {}
---
apiVersion: v1
kind: Service
metadata:
  name: postgres-db
spec:
  selector:
    app: postgresql-db
  type: LoadBalancer
  ports:
  - port: 5432
    targetPort: 5432
```

Apply the changes:

```
oc apply -f example-postgres.yaml
```

2.1.1.2. Hostname

For a production ready installation, you need a hostname that can be used to contact Red Hat build of Keycloak. See Configuring the hostname for the available configurations.

For development purposes, this chapter will use **test.keycloak.org**.

When running on OpenShift, with ingress enabled, and with the spec.ingress.classname set to openshift-default, you may leave the spec.hostname.hostname unpopulated in the Keycloak CR. The operator will assign a default hostname to the stored version of the CR similar to what would be created by an OpenShift Route without an explicit host - that is ingress-namespace.appsDomain If the appsDomain changes, or should you need a different hostname for any reason, then update the Keycloak CR.

2.1.1.3. TLS Certificate and key

See your Certification Authority to obtain the certificate and the key.

For development purposes, you can enter this command to obtain a self-signed certificate:

```
openssl req -subj '/CN=test.keycloak.org/O=Test Keycloak./C=US' -newkey rsa:2048 -nodes -keyout key.pem -x509 -days 365 -out certificate.pem
```

You should install it in the cluster namespace as a Secret by entering this command:

```
oc create secret tls example-tls-secret --cert certificate.pem --key key.pem
```

2.1.2. Deploying Red Hat build of Keycloak

To deploy Red Hat build of Keycloak, you create a Custom Resource (CR) based on the Keycloak Custom Resource Definition (CRD).

Consider storing the Database credentials in a separate Secret. Enter the following commands:

```
oc create secret generic keycloak-db-secret \
  --from-literal=username=[your_database_username] \
  --from-literal=password=[your_database_password]
```

You can customize several fields using the Keycloak CRD. For a basic deployment, you can stick to the following approach:

Create YAML file **example-kc.yaml**:

```
apiVersion: k8s.keycloak.org/v2alpha1
kind: Keycloak
metadata:
  name: example-kc
spec:
  instances: 1
  db:
    vendor: postgres
    host: postgres-db
    usernameSecret:
      name: keycloak-db-secret
      key: username
    passwordSecret:
      name: keycloak-db-secret
      key: password
  http:
    tlsSecret: example-tls-secret
  hostname:
    hostname: test.keycloak.org
  proxy:
    headers: xforwarded # double check your reverse proxy sets and overwrites the X-Forwarded-* headers
```

Apply the changes:

```
oc apply -f example-kc.yaml
```

To check that the Red Hat build of Keycloak instance has been provisioned in the cluster, check the status of the created CR by entering the following command:

```
oc get keycloaks/example-kc -o go-template='{{range .status.conditions}}CONDITION: {{.type}}{{"\n"}}
STATUS: {{.status}}{{"\n"}}  MESSAGE: {{.message}}{{"\n"}}{{end}}'
```

When the deployment is ready, look for output similar to the following:

```
CONDITION: Ready
  STATUS: true
  MESSAGE:
```

```
CONDITION: HasErrors
  STATUS: false
  MESSAGE:
CONDITION: RollingUpdate
  STATUS: false
  MESSAGE:
```

2.1.3. Accessing the Red Hat build of Keycloak deployment

The Red Hat build of Keycloak deployment is exposed through a basic Ingress and is accessible through the provided hostname. On installations with multiple default IngressClass instances or when running on OpenShift 4.12+ you should provide an ingressClassName by setting **ingress** spec with **className** property to the desired class name:

Edit YAML file **example-kc.yaml**:

```
apiVersion: k8s.keycloak.org/v2alpha1
kind: Keycloak
metadata:
  name: example-kc
spec:
    ...
    ingress:
      className: openshift-default
```

If the default ingress does not fit your use case, disable it by setting **ingress** spec with **enabled** property to **false** value:

Edit YAML file **example-kc.yaml**:

```
apiVersion: k8s.keycloak.org/v2alpha1
kind: Keycloak
metadata:
  name: example-kc
spec:
    ...
    ingress:
      enabled: false
```

Apply the changes:

```
oc apply -f example-kc.yaml
```

You can provide an alternative ingress resource pointing to the service **<keycloak-cr-name>-service**.

For debugging and development purposes, consider directly connecting to the Red Hat build of Keycloak service using a port forward. For example, enter this command:

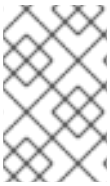
```
oc port-forward service/example-kc-service 8443:8443
```

2.1.3.1. Configuring the reverse proxy settings matching your Ingress Controller

The Operator supports configuring which of the reverse proxy headers should be accepted by server, which includes **Forwarded** and **X-Forwarded-*** headers.

If you Ingress implementation sets and overwrites either **Forwarded** or **X-Forwarded-*** headers, you can reflect that in the Keycloak CR as follows:

```
apiVersion: k8s.keycloak.org/v2alpha1
kind: Keycloak
metadata:
  name: example-kc
spec:
  ...
  proxy:
    headers: forwarded|xforwarded
```



NOTE

If the **proxy.headers** field is not specified, the Operator falls back to legacy behaviour by implicitly setting **proxy=passthrough** by default. This results in deprecation warnings in the server log. This fallback will be removed in a future release.

WARNING

When using the **proxy.headers** field, make sure your Ingress properly sets and overwrites the **Forwarded** or **X-Forwarded-*** headers respectively. To set these headers, consult the documentation for your Ingress Controller. Consider configuring it for either reencrypt or edge TLS termination as passthrough TLS doesn't allow the Ingress to modify the requests headers. Misconfiguration will leave Red Hat build of Keycloak exposed to security vulnerabilities.

For more details refer to the Using a reverse proxy guide.

2.1.4. Accessing the Admin Console

When deploying Red Hat build of Keycloak, the operator generates an arbitrary initial admin **username** and **password** and stores those credentials as a basic-auth Secret object in the same namespace as the CR.

WARNING

Change the default admin credentials and enable MFA in Red Hat build of Keycloak before going to production.

To fetch the initial admin credentials, you have to read and decode the Secret. The Secret name is derived from the Keycloak CR name plus the fixed suffix **-initial-admin**. To get the username and password for the **example-kc** CR, enter the following commands:

```
oc get secret example-kc-initial-admin -o jsonpath='{.data.username}' | base64 --decode
oc get secret example-kc-initial-admin -o jsonpath='{.data.password}' | base64 --decode
```

You can use those credentials to access the Admin Console or the Admin REST API.

CHAPTER 3. RED HAT BUILD OF KEYCLOAK REALM IMPORT

3.1. IMPORTING A RED HAT BUILD OF KEYCLOAK REALM

Using the Red Hat build of Keycloak Operator, you can perform a realm import for the Keycloak Deployment.

NOTE

- If a Realm with the same name already exists in Red Hat build of Keycloak, it will not be overwritten.
- The Realm Import CR only supports creation of new realms and does not update or delete those. Changes to the realm performed directly on Red Hat build of Keycloak are not synced back in the CR.

3.1.1. Creating a Realm Import Custom Resource

The following is an example of a Realm Import Custom Resource (CR):

```
apiVersion: k8s.keycloak.org/v2alpha1
kind: KeycloakRealmImport
metadata:
  name: my-realm-kc
spec:
  keycloakCRName: <name of the keycloak CR>
  realm:
    ...
```

This CR should be created in the same namespace as the Keycloak Deployment CR, defined in the field **keycloakCRName**. The **realm** field accepts a full RealmRepresentation.

The recommended way to obtain a **RealmRepresentation** is by leveraging the export functionality Importing and Exporting Realms .

1. Export the Realm to a single file.
2. Convert the JSON file to YAML.
3. Copy and paste the obtained YAML file as body for the **realm** key, making sure the indentation is correct.

3.1.2. Applying the Realm Import CR

Use **oc** to create the CR in the correct cluster namespace:

Create YAML file **example-realm-import.yaml**:

```
apiVersion: k8s.keycloak.org/v2alpha1
kind: KeycloakRealmImport
metadata:
  name: my-realm-kc
spec:
  keycloakCRName: <name of the keycloak CR>
```

```
  realm:
    id: example-realm
    realm: example-realm
    displayName: ExampleRealm
    enabled: true
```

Apply the changes:

```
oc apply -f example-realm-import.yaml
```

To check the status of the running import, enter the following command:

```
oc get keycloakrealmimports/my-realm-kc -o go-template='{{range .status.conditions}}CONDITION:
{{.type}}{{"\n"}}  STATUS: {{.status}}{{"\n"}}  MESSAGE: {{.message}}{{"\n"}}{{end}}'
```

When the import has successfully completed, the output will look like the following example:

```
CONDITION: Done
  STATUS: true
  MESSAGE:
CONDITION: Started
  STATUS: false
  MESSAGE:
CONDITION: HasErrors
  STATUS: false
  MESSAGE:
```

CHAPTER 4. ADVANCED CONFIGURATION

4.1. ADVANCED CONFIGURATION

This chapter describes how to use Custom Resources (CRs) for advanced configuration of your Red Hat build of Keycloak deployment.

4.1.1. Server configuration details

Many server options are exposed as first-class citizen fields in the Keycloak CR. The structure of the CR is based on the configuration structure of Red Hat build of Keycloak. For example, to configure the **https-port** of the server, follow a similar pattern in the CR and use the **httpsPort** field. The following example is a complex server configuration; however, it illustrates the relationship between server options and the Keycloak CR:

```
apiVersion: k8s.keycloak.org/v2alpha1
kind: Keycloak
metadata:
  name: example-kc
spec:
  db:
    vendor: postgres
    usernameSecret:
      name: usernameSecret
      key: usernameSecretKey
    passwordSecret:
      name: passwordSecret
      key: passwordSecretKey
    host: host
    database: database
    port: 123
    schema: schema
    poolInitialSize: 1
    poolMinSize: 2
    poolMaxSize: 3
  http:
    httpEnabled: true
    httpPort: 8180
    httpsPort: 8543
    tlsSecret: my-tls-secret
  hostname:
    hostname: my-hostname
    admin: my-admin-hostname
    strict: false
    strictBackchannel: false
  features:
    enabled:
      - docker
      - authorization
    disabled:
      - admin
      - step-up-authentication
  transaction:
    xaEnabled: false
```

For a list of options, see the Keycloak CRD. For details on configuring options, see All configuration.

4.1.1.1. Additional options

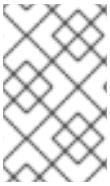
Some expert server options are unavailable as dedicated fields in the Keycloak CR. The following are examples of omitted fields:

- Fields that require deep understanding of the underlying Red Hat build of Keycloak implementation
- Fields that are not relevant to an OpenShift environment
- Fields for provider configuration because they are dynamic based on the used provider implementation

The **additionalOptions** field of the Keycloak CR enables Red Hat build of Keycloak to accept any available configuration in the form of key-value pairs. You can use this field to include any option that is omitted in the Keycloak CR. For details on configuring options, see All configuration.

The values can be expressed as plain text strings or Secret object references as shown in this example:

```
apiVersion: k8s.keycloak.org/v2alpha1
kind: Keycloak
metadata:
  name: example-kc
spec:
  ...
  additionalOptions:
    - name: spi-connections-http-client-default-connection-pool-size
      secret: # Secret reference
        name: http-client-secret # name of the Secret
        key: poolSize # name of the Key in the Secret
    - name: spi-email-template-mycustomprovider-enabled
      value: true # plain text value
```



NOTE

The name format of options defined in this way is identical to the key format of options specified in the configuration file. For details on various configuration formats, see Configuring Red Hat build of Keycloak .

4.1.2. Secret References

Secret References are used by some dedicated options in the Keycloak CR, such as **tlsSecret**, or as a value in **additionalOptions**.

Similarly ConfigMap References are used by options such as the **configMapFile**.

When specifying a Secret or ConfigMap Reference, make sure that a Secret or ConfigMap containing the referenced keys is present in the same namespace as the CR referencing it.

The operator will poll approximately every minute for changes to referenced Secrets or ConfigMaps. When a meaningful change is detected, the Operator performs a rolling restart of the Red Hat build of Keycloak Deployment to pick up the changes.

4.1.3. Unsupported features

The **unsupported** field of the CR contains highly experimental configuration options that are not completely tested and are Tech Preview.

4.1.3.1. Pod Template

The Pod Template is a raw API representation that is used for the Deployment Template. This field is a temporary workaround in case no supported field exists at the top level of the CR for your use case.

The Operator merges the fields of the provided template with the values generated by the Operator for the specific Deployment. With this feature, you have access to a high level of customizations. However, no guarantee exists that the Deployment will work as expected.

The following example illustrates injecting labels, annotations, volumes, and volume mounts:

```
apiVersion: k8s.keycloak.org/v2alpha1
kind: Keycloak
metadata:
  name: example-kc
spec:
  ...
  unsupported:
    podTemplate:
      metadata:
        labels:
          my-label: "keycloak"
      spec:
        containers:
          - volumeMounts:
              - name: test-volume
                mountPath: /mnt/test
        volumes:
          - name: test-volume
            secret:
              secretName: keycloak-additional-secret
```

4.1.4. Disabling required options

Red Hat build of Keycloak and the Red Hat build of Keycloak Operator provide the best production-ready experience with security in mind. However, during the development phase, you can disable key security features.

Specifically, you can disable the hostname and TLS as shown in the following example:

```
apiVersion: k8s.keycloak.org/v2alpha1
kind: Keycloak
metadata:
  name: example-kc
spec:
  ...
  http:
    httpEnabled: true
```

```
  hostname:
    strict: false
    strictBackchannel: false
```

4.1.5. Resource requirements

The Keycloak CR allows specifying the **resources** options for managing compute resources for the Red Hat build of Keycloak container. It provides the ability to request and limit resources independently for the main Keycloak deployment via the Keycloak CR, and for the realm import Job via the Realm Import CR.

When no values are specified, the default **requests** memory is set to **1700MiB**, and the **limits** memory is set to **2GiB**. These values were chosen based on a deeper analysis of Red Hat build of Keycloak memory management.

If no values are specified in the Realm Import CR, it falls back to the values specified in the Keycloak CR, or to the defaults as defined above.

You can specify your custom values based on your requirements as follows:

```
apiVersion: k8s.keycloak.org/v2alpha1
kind: Keycloak
metadata:
  name: example-kc
spec:
  ...
  resources:
    requests:
      cpu: 1200m
      memory: 896Mi
    limits:
      cpu: 6
      memory: 3Gi
```

Moreover, the Red Hat build of Keycloak container manages the heap size more effectively by providing relative values for the heap size. It is achieved by providing certain JVM options.

For more details, see Running Red Hat build of Keycloak in a container .

4.1.6. Truststores

If you need to provide trusted certificates, the Keycloak CR provides a top level feature for configuring the server's truststore as discussed in Configuring trusted certificates.

Use the truststores stanza of the Keycloak spec to specify Secrets containing PEM encoded files, or PKCS12 files with extension **.p12** or **.pfx**, for example:

```
apiVersion: k8s.keycloak.org/v2alpha1
kind: Keycloak
metadata:
  name: example-kc
spec:
  ...
  truststores:
```

```
    my-truststore:
      secret:
        name: my-secret
```

Where the contents of my-secret could be a PEM file, for example:

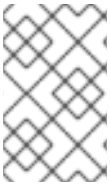
```
apiVersion: v1
kind: Secret
metadata:
  name: my-secret
stringData:
  cert.pem: |
    -----BEGIN CERTIFICATE-----
    ...
```

When running on a Kubernetes or OpenShift environment well-known locations of trusted certificates are included automatically. This includes /var/run/secrets/kubernetes.io/serviceaccount/ca.crt and the /var/run/secrets/kubernetes.io/serviceaccount/service-ca.crt when present.

CHAPTER 5. USING CUSTOM RED HAT BUILD OF KEYCLOAK IMAGES

5.1. RED HAT BUILD OF KEYCLOAK CUSTOM IMAGE WITH THE OPERATOR

With the Keycloak Custom Resource (CR), you can specify a custom container image for the Red Hat build of Keycloak server.



NOTE

To ensure full compatibility of Operator and Operand, make sure that the version of Red Hat build of Keycloak release used in the custom image is aligned with the version of the operator.

5.1.1. Best practice

When using the default Red Hat build of Keycloak image, the server will perform a costly re-augmentation every time a Pod starts. To avoid this delay, you can provide a custom image with the augmentation built-in from the build time of the image.

With a custom image, you can also specify the Keycloak *build-time* configurations and extensions during the build of the container.

For instructions on how to build such an image, see Running Red Hat build of Keycloak in a container .

5.1.2. Providing a custom Red Hat build of Keycloak image

To provide a custom image, you define the **image** field in the Keycloak CR as shown in this example:

```
apiVersion: k8s.keycloak.org/v2alpha1
kind: Keycloak
metadata:
  name: example-kc
spec:
  instances: 1
  image: quay.io/my-company/my-keycloak:latest
  http:
    tlsSecret: example-tls-secret
  hostname:
    hostname: test.keycloak.org
```

NOTE

With custom images, every build time option passed either through a dedicated field or the **additionalOptions** is ignored.

5.1.3. Non-optimized custom image

While it is considered a best practice use a pre-augmented image, if you want to use a non-optimized custom image or build time properties with an augmented image that is still possible. You just need set the **startOptimzed** field to **false** as shown in this example:

```
apiVersion: k8s.keycloak.org/v2alpha1
kind: Keycloak
metadata:
  name: example-kc
spec:
  instances: 1
  image: quay.io/my-company/my-keycloak:latest
  startOptimized: false
  http:
    tlsSecret: example-tls-secret
  hostname:
    hostname: test.keycloak.org
```

Keep in mind this will incur the re-augmentation cost on every start.