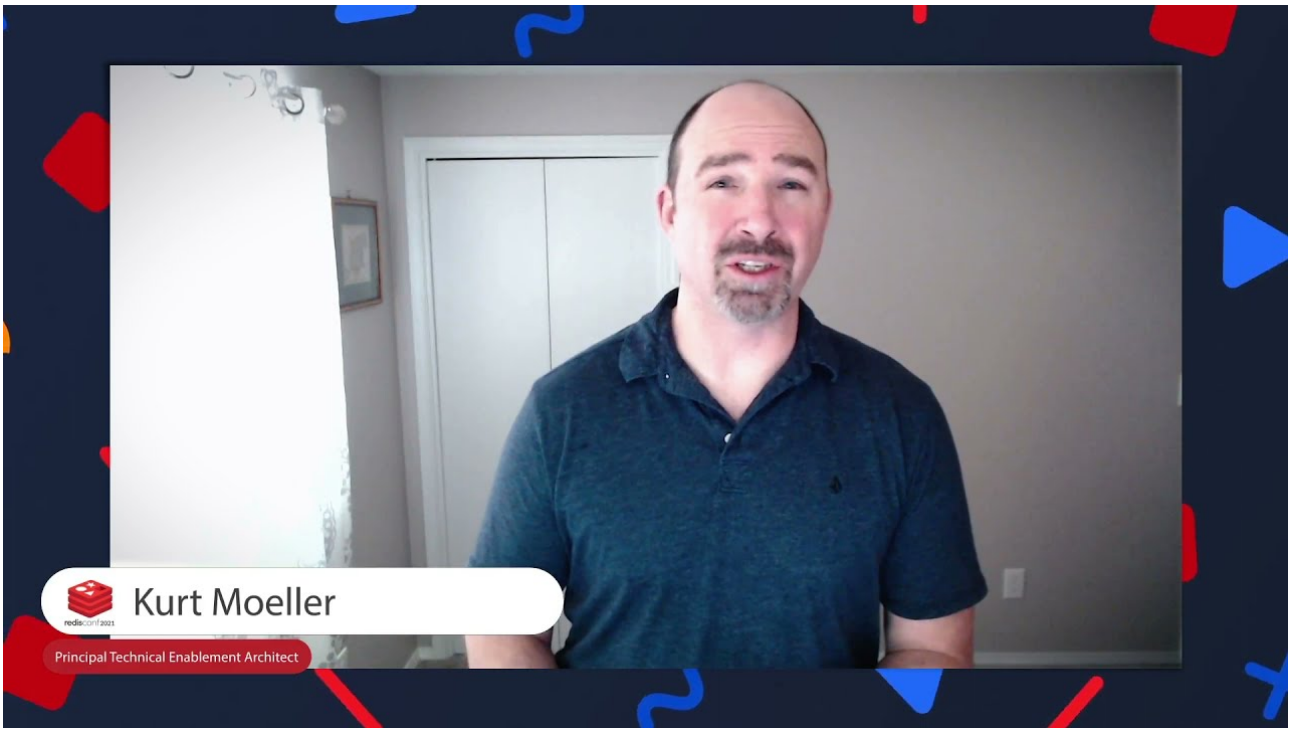
Unit 0

Running Redis at Scale

Course Introduction

0.1 Authors Introduction



YouTube URL

0.2 Prerequisites

- Access to a Linux-based system and familiarity with it
- Redis server and redis-cli installed (examples and exercises assume redis-server is in the $PATH)
- docker and docker-compose installed

- A git client and access to clone repos in Github. Some exercises will come from the following repository: https://github.com/redislabs-training/exercises-scaling-redis

0.3 Assumptions

- Comfortable with Linux Bash shell exercises
- Legacy terminology in Redis uses 'master' and 'slave' but in the course we will use 'primary' and 'replica.' You will still see the legacy terms in many commands, configurations, and field names.
- We will use $ to indicate command line prompt and > to indicate a redis-cli prompt

0.4 Course Overview

This course is broken up into units covering topics around scaling Redis for production deployment.

Scaling means more than just performance. We have tried to identify key topics that will help you have a performant, stable, and secure deployment of Redis.
This course is divided into the following units:

1. **Talking to Redis**: connection management and tuning of Redis.
2. **Securing the Connection and Data**: setting up TLS and access controls in Redis.
3. **Persistence/Durability**: options persisting Redus data to disk.
4. **High Availability**: how to make sure Redis and your data is always there.
5. **Scalability**: scaling Redis for both for higher throughput and capacity.

6. **Observability**: Visibility into your Redis deployment (metrics, etc.).

As you may know, Redis Labs leads and sponsors the development of open source Redis. We also provide Redis Enterprise as both an on-premises software deployment and a fully managed cloud service. We'll always support open source Redis, but we truly believe that Redis Enterprise provides the best experience for those running Redis in production.

As such, we'll focus the beginning of each chapter around open source Redis solutions. But we'll also cover the capabilities of Redis Enterprise and point out how it differs from open source Redis, where applicable.

For example, in the High Availability unit, we'll explore how to enable replication and we'll learn about Redis Sentinel. But we'll also look at Redis Enterprise's high availability features.

Our goal is to give you all the information you need to run Redis at scale, in whichever way is best for your organization. We hope you enjoy the course, and please don't hesitate to reach out on the course Discord channel if you have any questions along the way.

Sincerely,

Elena Kolevska
Redis Labs Technical Enablement Architect

Kurt Moeller
Redis Labs Technical Enablement Architect

About Redis Labs

Redis Labs, the main sponsor and contributor to Redis, developed Redis Enterprise as a DBaaS in the cloud 10 years ago. It has the

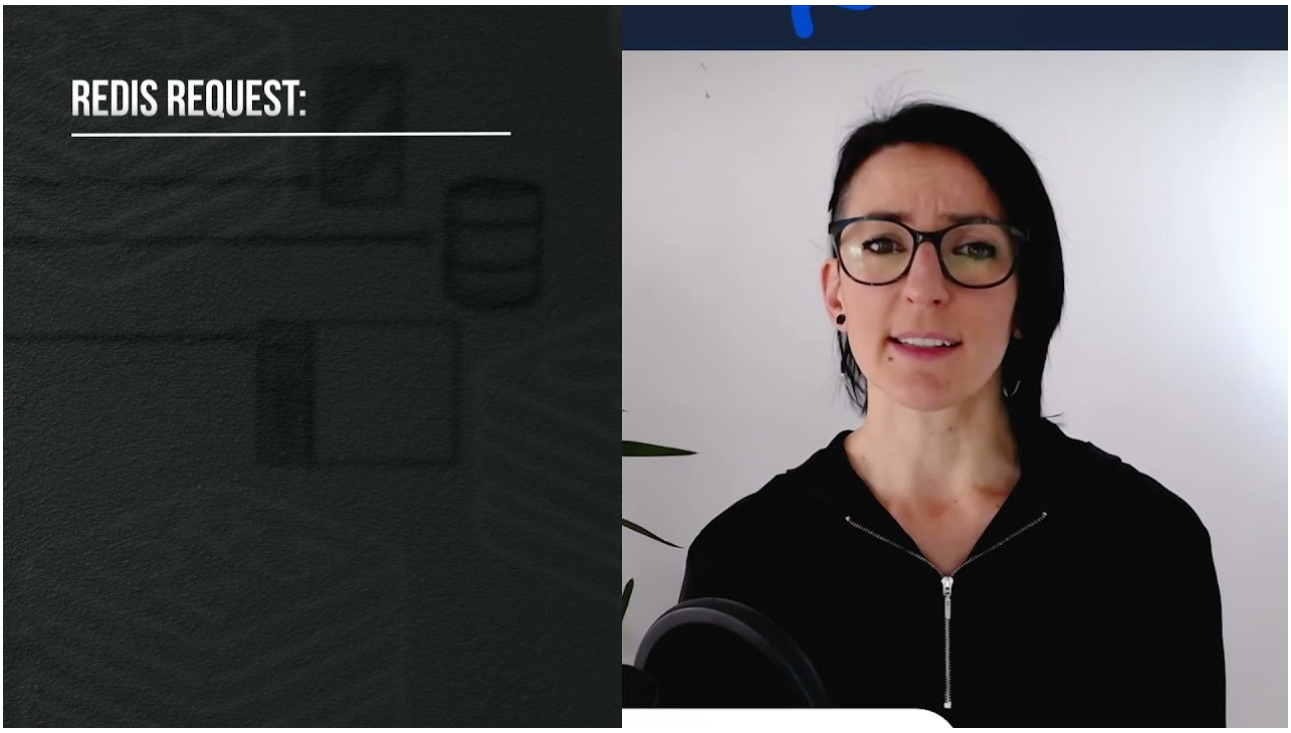
critical features you are going to want when deploying Redis at scale. In addition to the cloud deployment options on all the major providers (AWS, GCP and Azure), Redis Enterprise can be installed via a Kubernetes operator (across many flavors) or just downloaded and installed into the environment of your choice.

Unit 1

Course: Running Redis at Scale

Unit 1: Talking to Redis

1.1 Redis server overview



YouTube URL

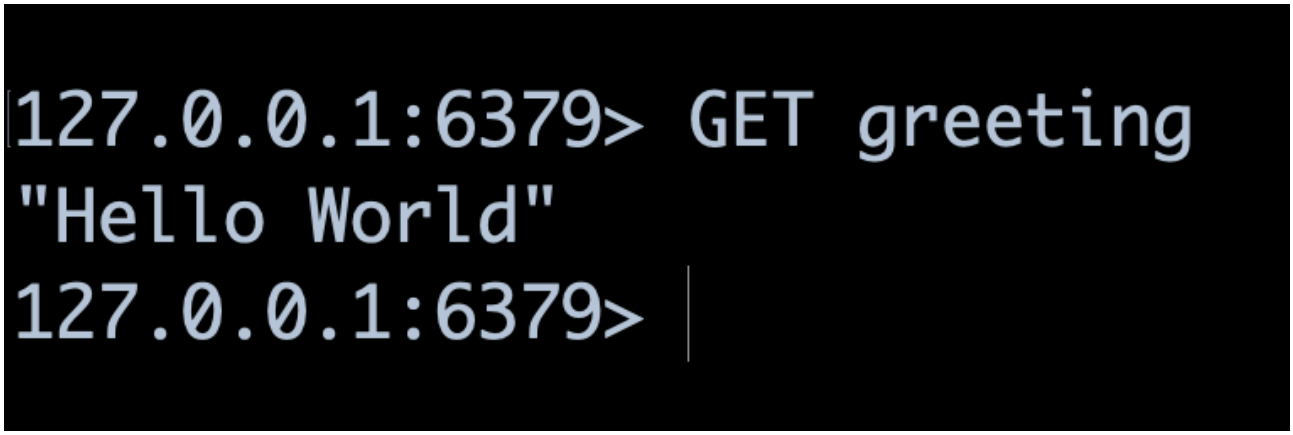
As you might already know, Redis is an open source data structure server written in C. You can store multiple data types, like strings, hashes, and streams and access them by a unique key name.

For example, if you have a string value “Hello World” saved under the key name “greeting”, you can access it by running:

```
GET greeting
```

All keys in a Redis database are stored in a flat keyspace. There is no enforced schema or naming policy, and the responsibility for organising the keyspace is left to the developer.

This is how the above command would look like in redis-cli - the command line interface for Redis:



The speed Redis is famous for is mostly due to the fact that Redis stores and serves data entirely from RAM memory instead of disk, as most other databases do.

Another contributing factor is its predominantly single-threaded nature: by using single threads, the server avoids race conditions and CPU-heavy context switching between the threads.
Indeed, this means that open source Redis can't take advantage of the processing power of multiple CPU cores, although CPU is rarely the bottleneck with Redis. You are more likely to bump up against memory or network limitations before hitting any CPU limitations.

That said, Redis Enterprise does let you take advantage of all of the threads on a single machine. More on that later.

Redis Requests

Let's now look at exactly what happens behind the scenes with every Redis request.

When a client sends a request to a Redis server, the request is first read from the socket, then parsed and processed and finally, the response is written back to the socket and sent to the user.

The reading and especially writing to a socket are expensive operations, so in Redis version 6.0 multi-threaded I/O was introduced.

When this feature is enabled, Redis can delegate the time spent reading and writing to I/O sockets over to other threads, freeing up cycles for storing and retrieving data and boosting overall performance by up to a factor of two.

Throughout the rest of the chapter, you'll learn how to use the Redis command line interface, how to configure your Redis server, and how to choose and tune your Redis client library.

1.2 The command-line tool: redis-cli

Redis provides a tool to send commands to the server and read its replies directly in the command line interface called redis-cli. It comes included in the Redis github repository and is automatically compiled when you build Redis from source.

You can run redis-cli in two modes: an **interactive mode** where the user types commands and sees the replies; and another mode where the command is sent as an argument to redis-cli, executed, and printed on the standard output.

redis-cli in interactive mode:

```
$ redis-cli
localhost:6379> SET foo bar
OK
localhost:6379> GET foo
"bar"
```

redis-cli in “command as an argument” mode:

```
$ redis-cli SET foo bar
OK
$ redis-cli GET foo
"bar"
```

Connecting to a database

Let’s use the CLI to connect to a Redis server running at 172.22.0.3 and port 7000:

```
$ redis-cli -h 172.22.0.3 -p 7000

172.22.0.3:7000>
```

The arguments -h and -p are used to specify the host and port, and they can be omitted if the server is running on localhost and the default port 6379.

Three tips: command history, hints, and auto-completion

The redis-cli provides some useful features for a more comfortable experience. For example, you can access your command history by pressing the arrow keys (up and down). You can also use the TAB key to autocomplete a command, saving even more keystrokes. Just type the first few letters of a command and keep pressing TAB until the command you want appears on screen.

Once you have the command name you want, the CLI will display syntax hints about the arguments so you don’t have to remember all of them, or open up the Redis command documentation.

These three tips can save you a lot of time and take you a step closer to being a power user.

--- Pro Tip ---

You can do much more with redis-cli, like sending output to a file, scanning for big keys, get continuous stats, monitor commands and

so on. For a much more detailed explanation refer to the documentation: https://redis.io/topics/rediscli .

1.3 Configuring a Redis server

The redis.conf file

The self-documented Redis configuration file, called **redis.conf**(sounds familiar?), is where you configure Redis. In this file, you can find all possible Redis configuration directives, together with a detailed description of what they do and their default values.

When running Redis in production, you should always adjust this file to your needs and instruct Redis to read its configuration from it. The way to do that is by providing the path to the file when starting up your server:

```
$ redis-server ./path/to/redis.conf
```

Passing arguments via the command line

When you're only starting a Redis server instance for testing purposes, you can pass configuration directives directly on the command line:

```
$ redis-server --port 7000 --replicaof 127.0.0.1 6379
```

The format of the arguments passed via the command line is exactly the same as the one used in the **redis.conf** file, except that each keyword is prefixed with --.
Note that, internally, this generates an in-memory temporary config file (possibly concatenating the config file passed by the user if any) where arguments are translated into the format of **redis.conf**.

Changing Redis configuration while the server is running

You can reconfigure a running Redis server without stopping or restarting it by using the special commands **CONFIG SET** and **CONFIG GET**.

Not all the configuration directives are supported in this way, but you can check the output of the command **CONFIG GET *** first for a list of all the supported ones.

Important: Modifying the configuration on the fly **has no effect on the redis.conf file,** so at the next restart of Redis, the old configuration will be used instead. If you want to force update the **redis.conf** file with your current configuration settings, you can run the CONFIG REWRITE command, which will automatically scan your redis.conf file and update the fields which don't match the current configuration value.

1.4 Redis clients

Redis has a **client-server architecture** and uses a **request-response model**. If your application issues the request, and the Redis server processes it, the **client library** acts as the messenger between the two. Client libraries perform a few duties:

1. **Parsing and encoding the messages the client and the server exchange**
2. **Making Redis commands idiomatic to your language**
3. **Managing the connection(s) to Redis**

Redis clients communicate with the Redis server over TCP, using a protocol called **RESP** (**RE**dis **S**erialization **P**rotocol) designed specifically for Redis. The default port on which they connect is **6379**, but this is, of course, configurable.

The **RESP** protocol is simple and text-based, so it is easily read by humans, as well as machines. A common request/response would look something like this. Note that we're using telnet here to send raw protocol:

```
> telnet 127.0.0.1 6379
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.


set greeting Hello
+OK
get greeting
$5
Hello
```

These characteristics, paired with the usual Redis-quality documentation of the protocol, make it really easy to create a new Redis client in your favourite language, but chances are, you won't have to do that because someone else has already written it; currently there are over 200 client libraries in over 50 languages listed on the redis.io/clients page.

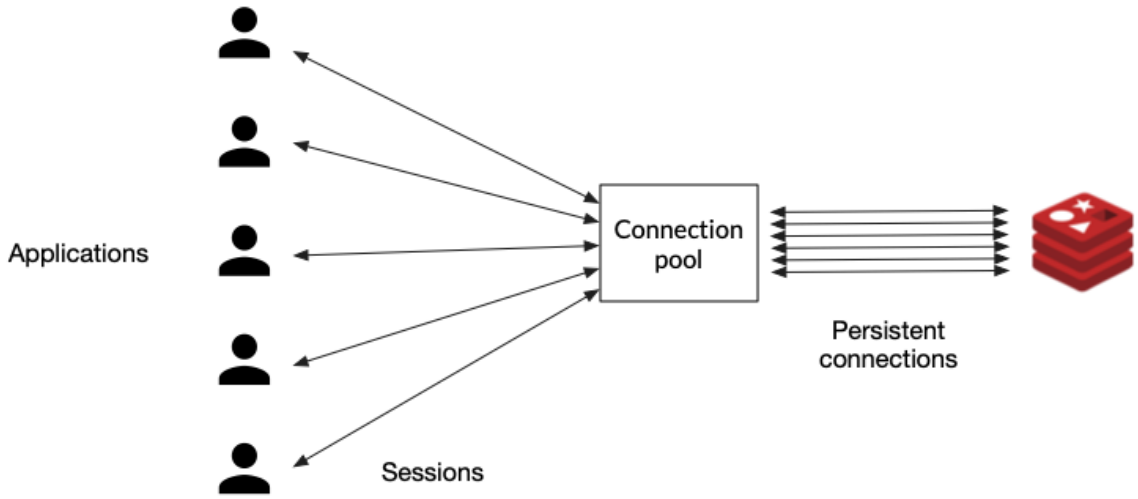
1.5 Client performance improvements

Connection management - Pooling

As we showed in the previous section, Redis clients are responsible for managing connections to the Redis server. Creating and recreating new connections over and over again, creates a lot of unnecessary load on the server. A good client library will offer some

way of optimising connection management, by setting up a connection pool, for example.

With connection pooling, the client library will instantiate a series of (persistent) connections to the Redis server and keep them open. When the application needs to send a request, the current thread will get one of these connections from the pool, use it, and return it when done.



So if possible, always try to choose a client library that supports pooling connections, because that decision alone can have a huge influence on your system's performance.

Pipelining

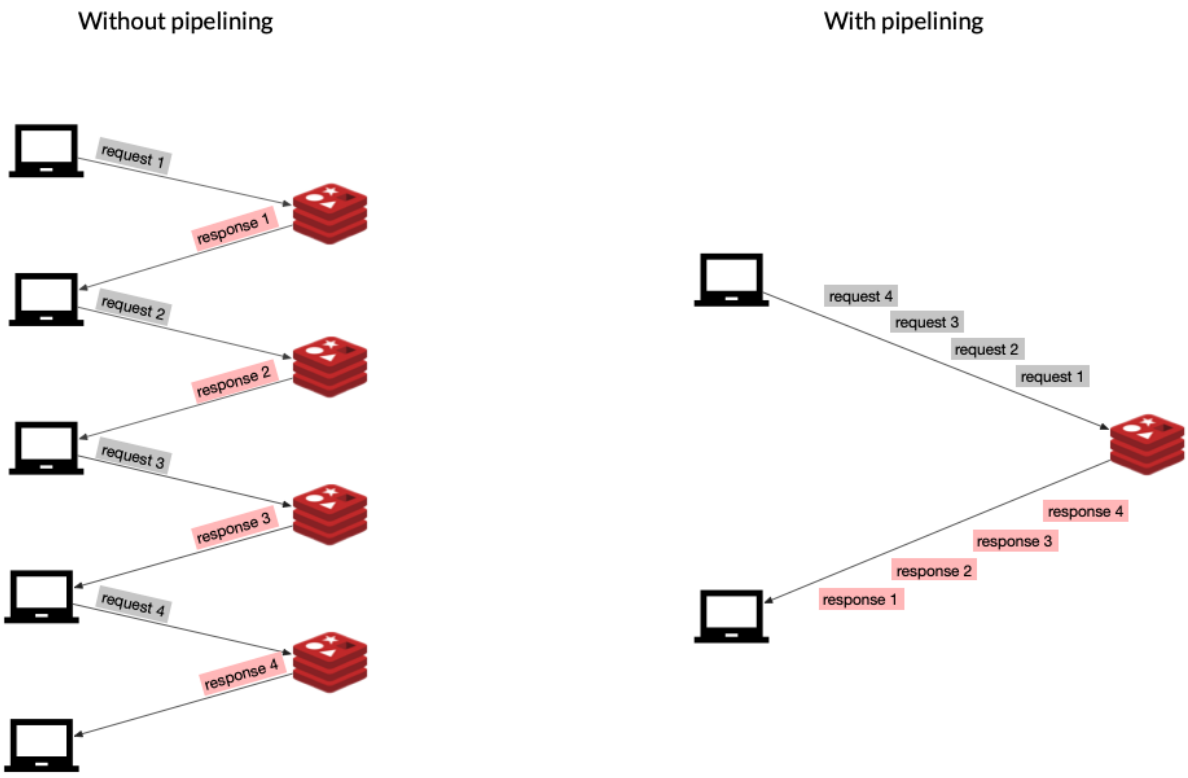
As in any client-server application, Redis can handle many clients simultaneously.

Each client does a (typically blocking) read on a socket and waits for the server response. The server reads the request from the socket, parses it, processes it, and writes the response to the socket. The time the data packets take to travel from the client to

the server, and then back again, is called **network round trip time** (RTT).

If, for example, you needed to execute 50 commands, you would have to send a request and wait for the response 50 times, paying the RTT cost every single time. To tackle this problem, Redis can process new requests even if the client hasn't already read the old responses. This way, you can send multiple commands to the server without waiting for the replies at all; the replies are read in the end, in a single step.

This technique is called pipelining and is another good way to improve the performance of your system. Most Redis libraries support this technique out of the box.



1.6 Initial Tuning

Intro to Tuning

We love Redis because it’s fast (and fun!), so as we begin to consider scaling out Redis, we first want to make sure we've done everything we can to maximize its performance.

Let's start by looking at some important tuning parameters.

Max Clients

Redis has a default of max of 10,000 clients; after that maximum has been reached, Redis will respond to all new connections with an error. If you have a lot of connections (or a lot of application instances), then you may need to go higher. You can set the max number of simultaneous clients in the Redis config file (redis.conf):

```
maxclients 20000
```

Max Memory

By default, Redis has no max memory limit, so it will use all available system memory. If you are using replication, you will want to limit the memory usage in order to have overhead for replica output buffers. It’s also a good idea to leave memory for the system. Something like 25% overhead. You can update this setting in Redis config file:

```
# memory size in bytes
maxmemory 1288490188
```

Set tcp-backlog

The Redis server uses the value of tcp-backlog to specify the size of the complete connection queue. Redis passes this configuration as the second parameter of the listen(int s, int backlog) call.

If you have many connections, you will need to set this higher than the default of 511. You can update this in Redis config file:

```
# TCP listen() backlog.
#
# In high requests-per-second environments you
need an high backlog in order
# to avoid slow clients connections issues. Note
that the Linux kernel
# will silently truncate it to the value of /proc/
sys/net/core/somaxconn so
# make sure to raise both the value of somaxconn
and tcp_max_syn_backlog
# in order to get the desired effect.
tcp-backlog 65536
```

As the comment in `redis.conf` indicates, the value of `somaxconn` and `tcp_max_syn_backlog` may need to be increased at the OS level as well.

Set read replica configurations

One simple way to scale Redis is to add read replicas and take load off of the primary. This is most effective when you have a ready-heavy (as opposed to write-heavy) workload. You will probably want to have the replica available and still serving stale data, even if the replication is not completed. You can update this in the Redis config:

```
slave-serve-stale-data yes
```

You will also want to prevent any writes from happening on the replicas. You can update this in the Redis config:

```
slave-read-only yes
```

Kernel memory

Under high load, occasional performance dips can occur due to memory allocation. This is something Salvatore, the creator of Redis, blogged about in the past. The performance issue is related to transparent hugepages, which you can disable at the OS level if needed.

```
echo 'never' > /sys/kernel/mm/
transparent_hugepage/enabled
```

Kernel network stack

If you plan on handling a large number of connections in a high performance environment, we recommend tuning the following kernel parameters:

```
vm.swappiness=0                         # turn off
swapping
net.ipv4.tcp_sack=1                     # enable
selective acknowledgements
net.ipv4.tcp_timestamps=1               # needed for
selective acknowledgements
net.ipv4.tcp_window_scaling=1           # scale the
network window
net.ipv4.tcp_congestion_control=cubic # better
congestion algorithm
net.ipv4.tcp_syncookies=1               # enable syn
cookies
net.ipv4.tcp_tw_recycle=1               # recycle
sockets quickly
net.ipv4.tcp_max_syn_backlog=NUMBER  # backlog
setting
net.core.somaxconn=NUMBER               # up the
number of connections per port
net.core.rmem_max=NUMBER                # up the
receive buffer size
```

```
net.core.wmem_max=NUMBER                # up the
buffer size for all connections
```

File descriptor limits

If you do not set the correct number of file descriptors for the Redis user, you will see errors indicating that “Redis can’t set maximum open files..” You can increase the file descriptor limit at the OS level.

Example on Ubuntu using systemd:

```
/etc/systemd/system/redis.service
[Service]
...
User=redis
Group=redis
...
LimitNOFILE=65536
...
```

Then you will need to reload the daemon and restart the redis.service.

Enabling RPS (Receive Packet Steering) and CPU preferences

One way we can improve performance is to prevent Redis from running on the same CPUs as those handling any network traffic. This can be accomplished by enabling RPS for our network interfaces and creating some CPU affinity for our Redis process.

Here is an example. First we can enable RPS on CPUs 0-1:

```
echo '3' > /sys/class/net/eth1/queues/rx-0/
rps_cpus
```

Then we can set the CPU affinity for redis to CPUs 2-8:

```
# config is set to write pid to /var/run/redis.pid
$ taskset -pc 2-8 `cat /var/run/redis.pid`
pid 8946's current affinity list: 0-8
pid 8946's new affinity list: 2-8
```

1.7 Quiz

Please click here to take the quiz.

Unit 2

Course: Running Redis at Scale

Unit 2: Securing Connections and Data

2.1 Introduction to securing Redis

Just like any other software deployment, as soon as you need to run Redis in production, security is going to come up. You should have policies in place to secure access to the systems that Redis will be running on.

You will want to secure data in transit through TLS and even add mutualTLS for client authentication. Redis ACLs provide a way to restrict access (and let you implement the principle of least privilege) by only granting access to the commands and keys that are needed.

Redis Enterprise adds Role Based Access Control, LDAP, and other advanced security features.

Let's now dive deeper into these security features and get hands-on with the exercises.

2.2 Basic security settings

Network and OS Level

You should take steps to secure the system and overall network context where you'll be deploying Redis. As redis.io states concerning the Redis security model:

“Redis is designed to be accessed by trusted clients inside trusted environments. This means that usually it is not a good idea to expose the Redis instance directly to the internet or, in general, to an environment where untrusted clients can directly access the Redis TCP port or UNIX socket.”

Set up a firewall to prevent any access from any unauthorized sources. Deploy Redis where it is only available to internal traffic and not open to the internet where untrusted clients can attempt to access it.

Do not run Redis as root on the system, and make sure that only a specific user has access to the data directory folder. Encrypting the disks where data is stored provides some security around data at rest.

Unfortunately, getting into the details of network and OS-level security is beyond the scope of this course, and what you do depends a lot on the deployment environment. For this, it's best to consult with your organization's policies and a security professional.

Basic server configurations

port

The default Redis port is 6379. This well-known port is frequently the target of automated attacks. Unless you are targeted by a motivated attacker directly, modifying the Redis port can be a way to ensure that scripts run against systems searching for vulnerabilities do not target you.

During the TLS section, we will see how you can disable the default TCP port entirely and enable a TLS-only port.

bind

You can and usually should bind Redis to a single interface via the Redis configuration. This is the default:

```
bind 127.0.0.1 -::1
```

Here are some other examples included in the default Redis configuration.

```
# bind 192.168.1.100 10.0.0.1 # listens on two
specific IPv4 addresses
# bind 127.0.0.1 ::1          # listens on loopback
IPv4 and IPv6
# bind * -::*                 # all available
interfaces
```

Binding to all network interfaces is dangerous when Redis is exposed to the internet.

requirepass

You should set a default password, which will force clients to authenticate (using the AUTH command) before they can run any commands or get any data.

```
requirepass fTw&00g8*
```

We'll cover additional user and ACL features later in the course.

protected-mode

If the server is not binding to a set of addresses (like we showed above) and no password is configured (like we showed above), protected-mode will cause the server to only accept connections

from clients connecting from loopback addresses 127.0.0.1 and ::1 and from Unix domain sockets.

The default is enabled:

```
protected-mode yes
```

Disable protected mode only if you are sure you want clients from other hosts to connect to Redis even if no authentication is configured and no specific set of interfaces have been listed under the bind directive.

rename-commands

Another common security/safety step is to rename dangerous commands.

```
rename-command FLUSHALL "CUSTOMDELALL"
rename-command CONFIG "CUSTOMCONF"
```

Since ‘security through obscurity’ is known to be a weak (and typically unacceptable) strategy, it's better to use Redis ACLs to allowing or disallowing the dangerous command category for specific users. We'll cover that later.

2.3 Enabling TLS

TLS

Transport Layer Security, or TLS as it's commonly known, is a series of protocols that secure communications across a network. As you may already know, TLS is a critical tool in securing data in transit. The last thing we want with our Redis deployment is to be susceptible to a ‘man in the middle’ attack. Now let’s look at how we can enable TLS in Redis.

Building with TLS

The first step needed is to build Redis with TLS support.
`make BUILD_TLS=yes`

Configuring Port and Certificates

A TLS port must also be specified in addition to the server certificates and private key.

When enabling TLS, you will probably want to disable the ability to accept connections on an unencrypted socket. You can do this by setting the normal port configuration to 0.

You can configure TLS in Redis in two ways. The first is to simply pass the values in flags during startup:

```
./src/redis-server --tls-port 6443 --port 0 \
   --tls-cert-file /etc/ssl/certs/redis-server.crt
\
   --tls-key-file /etc/ssl/certs/redis-server.key
\
   --tls-ca-cert-file /etc/ssl/certs/ca.crt
```

This method is best when your just testing a TLS configuration.

For production deployments, you'll want to enable TLS by setting the appropriate directives in the Redis configuration file:

```
tls-cert-file /etc/ssl/certs/redis-server.crt
tls-key-file /etc/ssl/certs/redis-server.key
tls-ca-cert-file /etc/ssl/certs/ca.crt
```

This configuration will not utilize the system-wide configuration when looking to authenticate a client certificate against a certificate

authority. The examples above use the `tls-ca-cert-file`, but it is also possible to configure a CA certificates directory `tls-ca-cert-dir`, which will look at system-installed certificates

Client Connections

Redis can enforce mutual TLS (mTLS). mTLS requires the client to authenticate with a valid certificate. The client will need to pass a certificate which can be validated against the CA certificates that the server is configured for.

To connect to a Redis server with mTLS enabled, we can use redis-cli, passing in the client's public certificate and private key. We also provide a CA certificate to validate the server's certificate against.

```
redis-cli --tls --cert /etc/certs/client.crt --key
etc/certs/client.key --cacert /etc/certs/ca.crt
```

Not all environments require mTLS. If this level of authentication is not required, you can also disable client authentication Instead, you can enable 'one-way' TLS, which still ensures that your data is encrypted in transit.

```
tls-auth-clients no
```

Hardening TLS

Beyond the basic setup of your TLS server certificates, there are some options that may enable you to satisfy additional security requirements.

You can decide which TLS versions you wish to support. The default is currently to support TLS 1.2 and 1.3.

```
tls-protocols "TLSv1.2 TLSv1.3"
```

You can configure the ciphersuites to be used with TLSv1.2.

```
tls-ciphersuites TLS_CHACHA20_POLY1305_SHA256
```

Normally, the server will prefer the client's cipher preference, but this can be switched.

```
tls-prefer-server-ciphers yes
```

A TLS session cache is enabled by default to improve performance and cost of reconnections, but this can be disabled or tuned. There are a series of configurations related to tls-session.

```
#tls-session-caching no

tls-session-cache-size 4500

tls-session-cache-timeout 30
```

Additional Configurations

There are configurations for securing replication, cluster and sentinel that should be explored if those features are utilized in your deployment. Example:

```
tls-replication yes

tls-cluster yes
```

The Redis configuration file is well documented and an excellent source of information for each of these features.

2.4 Exercise - Enabling TLS

Clone this repo if you have not already: https://github.com/redislabs-training/exercises-scaling-redis

Change into the ‘security-tls’ directory.

Requirements

- docker
- docker-compose
- internet connection

Starting the Environment

```
docker-compose up -d
```

Connect to the Environment

In a terminal, run this command to get a shell prompt inside the running docker container:
`docker-compose exec redis_tls bash`

Container Context

The docker-compose will startup a docker container based on the standard Redis docker image, but it will create some self-signed certs so that you can try out the TLS support.

If you ls the `/etc/certs/` directory, you'll see the client and server certs.

If you wish to understand how these self-signed certs were created, you can view the shell script used either in this repo gen-certs.sh or inside the container `/tmp/gen-certs.sh` where it was copied. This was adapted from a similar script in the Redis repo under utilities that is used for testing.

Note: Using a container like this is best for a quick lightweight learning environment to help you understand how to enable and work with the TLS settings. It is not intended to demonstrate how to deploy Redis.

Starting up Redis with TLS flags

One way you can enable TLS support is by adding the TLS flags to the redis-server startup. You can enable the TLS port and disable the regular TCP port, point to the certificate, key and CA cert.

```
redis-server --tls-port 6379 --port 0 --tls-cert-file /etc/certs/server.crt --tls-key-file /etc/certs/server.key --tls-ca-cert-file /etc/certs/ca.crt
```

If you ctl-c to exit the running process the Redis server will also stop.

Starting up Redis with TLS configuration

The docker container has also had the default redis.conf file copied to `/usr/local/etc/redis/redis.conf`, but in the previous command we did not utilize it.
Let's try to enable the same TLS settings that we did above in this config and then start Redis with TLS support that way.

```
redis-server /usr/local/etc/redis/redis.conf
```

Now let's verify that we can connect using TLS from a client.

Connecting redis-cli using TLS

Open another terminal window or tab and run this command to get a shell prompt inside that same docker container:
`docker-compose exec redis_tls bash`

We can use redis-cli to connect to our TLS enabled Redis server.
`redis-cli --tls --cert /etc/certs/client.crt --key /etc/certs/client.key --cacert /etc/certs/ca.crt`

If you are able to run the `INFO` or other commands you know that your client certificates have been accepted and TLS is being used for the connection. If you are returned an error when trying to run a command you may need to:

- verify you have entered all the flags correctly
- go back to the first terminal window and check the errors
- fix any misconfigurations in the redis.conf and restart the server
- retest after making changes until you can run commands successfully

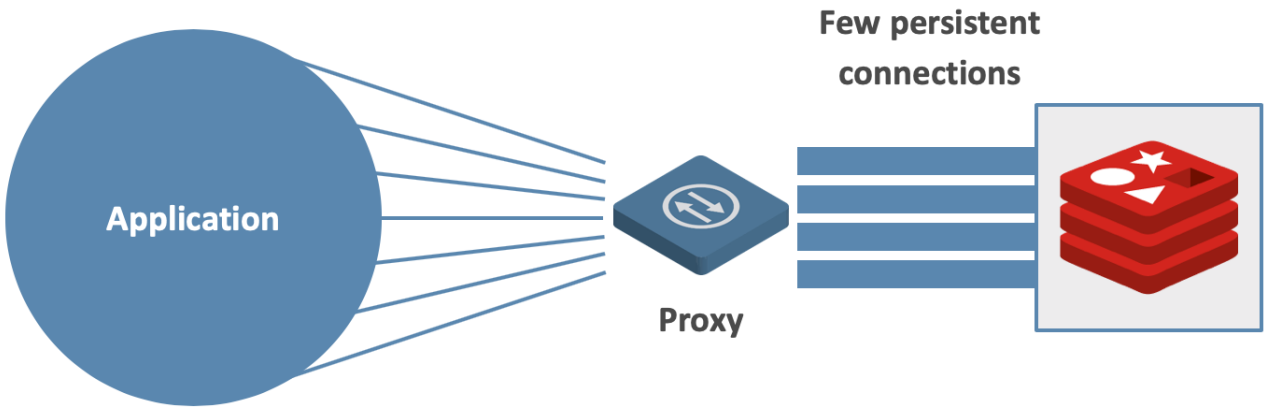
Stopping Environment

`docker-compose down`

2.5 Managing TLS on a Database in Redis Enterprise

Proxy Connection Handling

Redis Enterprise provides data access through a multi-threaded proxy process that manages and optimizes access to shards within the cluster. The proxy takes the expensive client connection handling off of the Redis server processes.

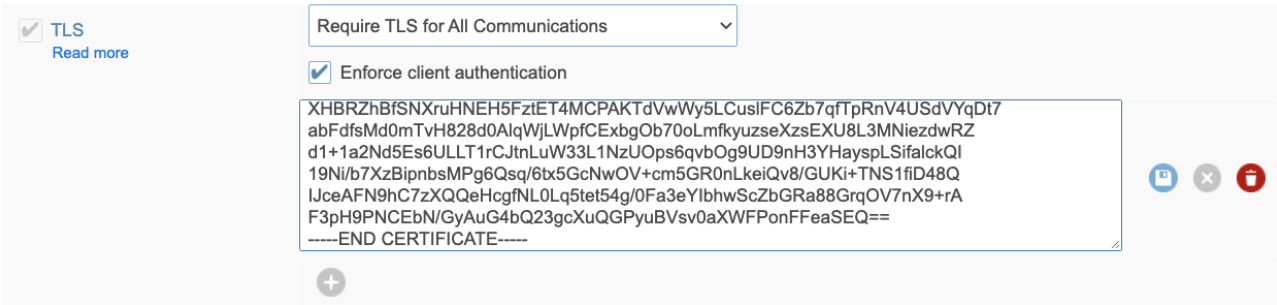


When TLS is enabled on a Redis Enterprise database, the proxy handles the TLS handshake. This means that TLS version checks and client authentication (if enabled) happen without impacting your Redis server process.

In addition to handling TLS, there are additional security benefits of having the proxy in front of your databases, such as being able to block commands with known security implications (e.g. CONFIG SET).

Adding TLS to a Database

In Redis Enterprise, you manage TLS using a built-in UI known as the admin console. UI. Here is a screenshot of the database configuration editor where you can enable TLS and client authentication.



There will be a few exercises in the course that will give you a chance to get hands-on with the Redis Enterprise admin console.

2.6 Access to Commands and Keys

Why ACLs?

Redis ACLs expand the basic password capability in Redis to support multiple users with separate usernames and passwords and to specify what commands and/or keys those users have access to. Administrators can specify access levels to commands or data.

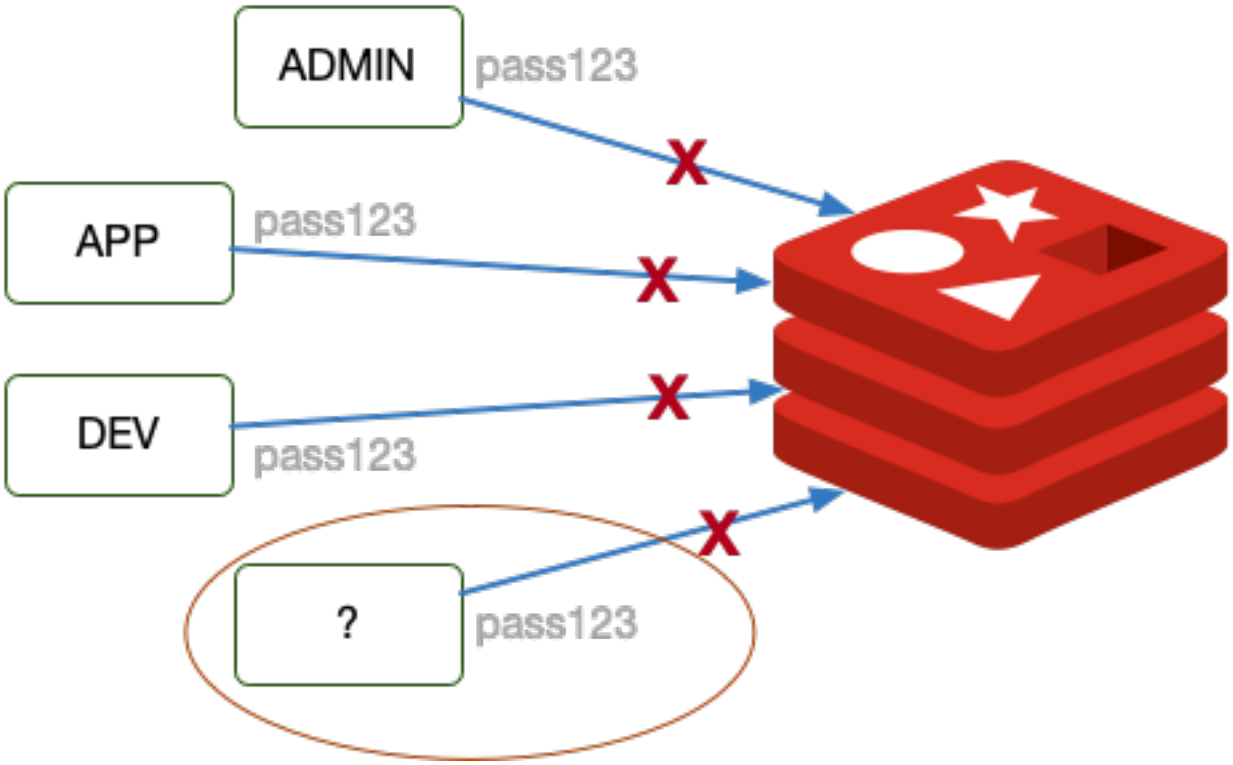
Restricting Access

- Prevents untrusted clients from accessing what they don't need
- Allows trusted clients to access do what they need
- Example: a client that is only subscribing to a notification published by another entity doesn't need access to full write commands

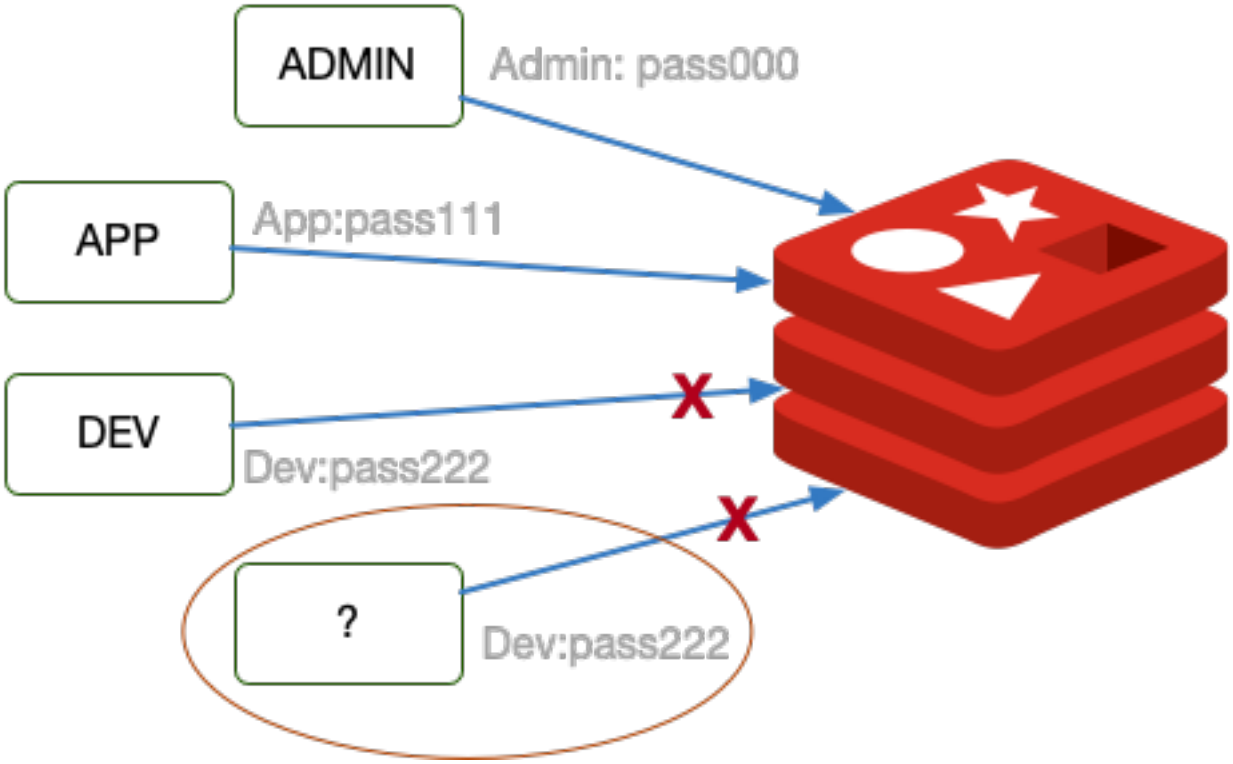
Operational Safety

- Limits administrative commands to appropriate personnel
- Limits destructive or dangerous commands
- Example: a worker fetching delayed jobs does not need access to `FLUSHALL`

Let’s look at an example of how ACLs can help in dealing with a bad actor accessing a Redis instance. If all of the clients accessing Redis are using the same password, it is not possible to revoke access specifically. We could change the password which would result in removing access to all clients.

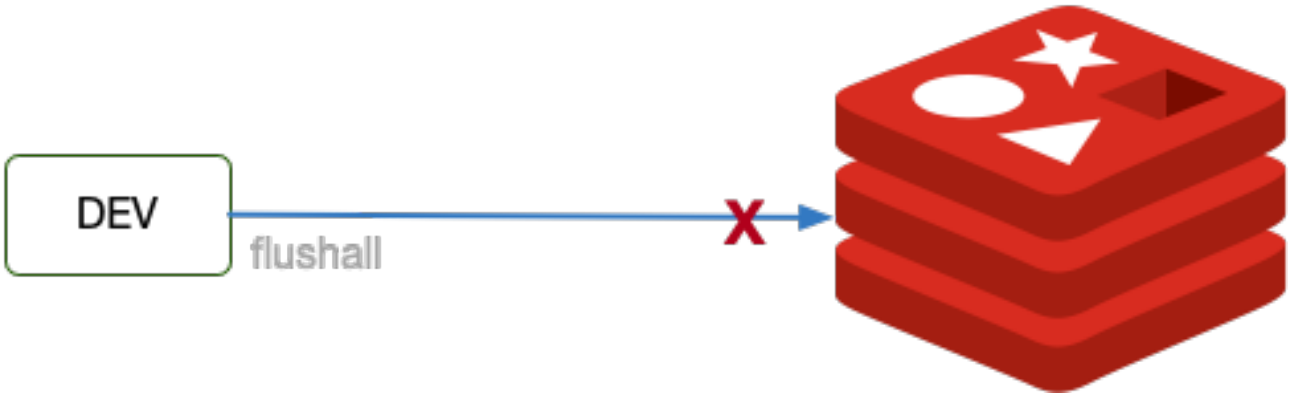


By assigning specific accounts to specific users, it's much easier to isolate the compromised account and block the bad actor without impacting all clients.

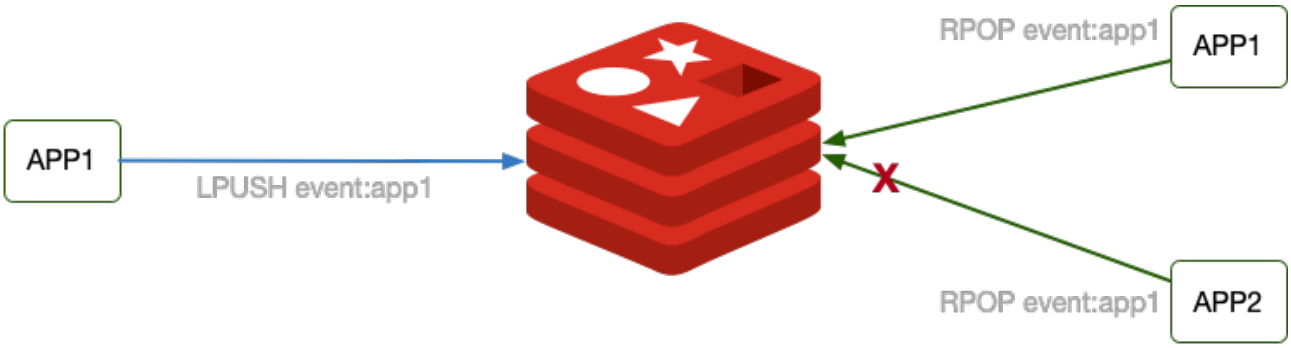


Beside being able to identify users more granularly, Redis ACLs let you grant or deny access to specific commands or command

categories. You can block a certain user from any ‘dangerous’ category of commands like `FLUSHALL`.



Users can also be granted access to specific keys or key patterns. You can grant/deny clients access to specific keys, like `event:app1` below, or use an expression to match multiple keys.



Now let’s explore some of the details around Redis ACLs.

Creating Users and ACLs

To create users you, run the ACL SETUSER command.

Syntax highlights:

- The user state can be on or off
- You can add a cleartext password with “>” remove with “<”
- You can add hashed password with "#" and remove with "!"
- Use 'nopass' for no password
- Permissions reset can be done with the 'reset' directive

- **Key restrictions** with ~ operator using "+" to add or "-" to restrict
- **Category restrictions** with @ operator using "+" to add or "-" to restrict

Creating an admin user:

```
> acl setuser admin on >redis123 ~* -@all +@admin
```

Once an admin user is created, you can safely disable the default password:

```
> acl setuser default off >defoff ~* -@all
```

To avoid passing the user password in plain text into the client[1] [2] [3] [4] , you can create a hash:

```
echo -n "redis123" | openssl dgst -sha256
cb8001ad8314aa0abd372e48210e44b1029975a84dbef6e64b
5de1f5054e3f4b
```

and then a user using that hash for the password:

> ACL SETUSER readapp on #cb8001ad8314aa0abd372e48210e44b1029975a84dbef6e64b5de1f5054e3f4b -@all +hget +get ~*

These capabilities allow you to create many different types of service accounts. For example, this account can run everything but the "dangerous" command category:

```
> acl setuser geoapp on >anotherPwd34 +@all
-@dangerous ~*
```

This account can only access the geo commands and keys (with ‘place:’ in the name):

```
> acl setuser geoapp on >anotherPwd34 ~place:*
-@all +@geo
```

If you need to figure out what the different ACL categories contain, Redis provides a way to list the categories and drill into the specific commands included in them.

List all the ACL categories:

```
> acl cat
1) "keyspace"
2) "read"
3) "write"
4) "set"
5) "sortedset"
6) "list"
...
```

List the commands in a specific category:

```
> acl cat admin
1) "monitor"
2) "replicaof"
3) "save"
4) "acl"
5) "bgrewriteaof"
6) "debug"
```

Users can be kept in memory or in a file on disk. There are few sub commands for managing this.

Load an ACL config from a file or save the existing ACL config on the server to a file

First Redis can be configured to use a specific ACL file in your redis configuration file:

```
aclfile /etc/redis/users.acl
```

If that file is modified it can be reloaded

```
> acl load
```

If you want to write current ACL configuration to the ACL file

```
> acl save
```

Authenticating Users

User's use the existing AUTH command to authenticate to Redis.

```
AUTH app anotherPwd34
```

User Management

As an admin there are a few ACL sub commands that allow you manage the users accessing Redis.

The DELUSER command gives you the ability to delete one or multiple accounts.

```
> ACL DELUSER readapp
```

You can get a list of users, full list of users and their ACLs.

In order to get a simple list of current users use the USERS sub command.

```
> acl users
1) "AppGeoService"
2) "Default user"
```

For a full list of users and ACLs use the LIST sub command.

```
> acl list
1) "user admin on
#cb8001ad8314aa0abd372e48210e44b1029975a84dbef6e64
```

```
b5de1f5054e3f4b ~* -@all +@admin +@dangerous
-restore-asking -flushdb -keys -migrate -restore
-sort -flushall -info -role -swapdb"
2) "user default on
#9e5e1a2d9f353394941e196efa358292b3557e795be116012
05cd9e48eebc1e7 ~* -@all"
3) "user newuser on
#41cbe1a87981490351ccad5346d96da0ac10678670b31fc0a
b209aed1b5bc515 -@all +get"
```

It is also possible to get details for a specific user using the GETUSER sub command:

```
> acl getuser admin
1) "flags"
2) 1) "on"
 2) "allkeys"
3) "passwords"
4) 1)
"cb8001ad8314aa0abd372e48210e44b1029975a84dbef6e64
b5de1f5054e3f4b"
5) "commands"
6) "-@all +@admin +@dangerous -restore-asking
-flushdb -keys -migrate -restore -sort -flushall
-info -role -swapdb"
7) "keys"
8) 1) "*"
```

Now that you have a good overview of how users and ACLs can be managed on Redis, let's get hands-on with this feature.

2.7 Exercise - Setting ACLs for commands and keys

Clone this repo if you have not already: https://github.com/redislabs-training/exercises-scaling-redis

Change into the ‘security-acls’ directory.

Requirements

- docker
- docker-compose
- internet connection

Starting the Environment

```
docker-compose up -d
```

Connect to the Environment

In a terminal run this command to get a redis-cli prompt inside the running docker container:
`docker-compose exec redis_acls redis-cli`

Update the default account

Using the ACL SETUSER command set the password for the default account to Sec&789. Afterwards you should be able to run this command and get back an OK response:

```
AUTH default Sec&789
```

If you get this error then you have set the password incorrectly:

```
(error) WRONGPASS invalid username-password pair
```

Use command category ACLs

1. View all categories of commands that can be used with ACLs:
2. `ACL CAT`
3. View the specific commands of a given category:
4. `ACL CAT dangerous`
5. Add a new user with a password and access to the hash category.
6. Authenticate using the new username and password using the AUTH command.

7. Verify proper access was granted A SET command should return the following error but the HSET command should not.
8. `(error) NOPERM this user has no permissions to run the 'set' command or its subcommand`
9. Remember to make sure that a key ACL was also set: ~* otherwise it won't work.

Use key-based ACL

1. Auth back to the default user
2. Set some keys:
3. `mset bucket:1 dirt bucket:2 turf pail:1 sand`
4. Add a new user `bucket-reader` that is enabled with a password redis123 with *read only access to keys starting with bucket:
5. Authenticate using the new username and password using the AUTH command.
6. Verify proper access was granted:
7. can get `bucket:1`
8. can NOT get `pail:1`
9. can NOT set `bucket:3` or any other key

Validation should look something like this:

```
> auth bucket-reader redis123
OK
> get bucket:1
"dirt"
> get pail:1
(error) NOPERM this user has no permissions to
access one of the keys used as arguments
> set bucket:3 water
(error) NOPERM this user has no permissions to run
the 'set' command or its subcommand
```

ACL admin and other utilities

1. Auth back to the default user
2. Run `ACL HELP` to view available ACL commands
3. Verify you are the default user by running `ACL WHOAMI`

4. Run `ACL LIST` to view all current users and ACLs
5. Run `ACL LOG` to view auth events
6. Run `ACL DELUSER` bucket-reader then `ACL LIST` to verify the user was deleted

Stopping Environment

```
docker-compose down
```

2.8 Role-Based Access Control in Redis Enterprise

Role-Based Access Control

Maintaining each user's access individually can create a lot of overhead and complexity for an organization. Role based access control (RBAC) allows you to assign appropriate roles to users granting them the privileges they need. This minimizes excessive permissioning and error-prone user-specific grants.

Redis Enterprise implements Role-Based Access Control (RBAC) for access to the admin console, REST API, and individual databases.

Users

- Username and Email identifier
- Assigned a role
- LDAP authentication an option

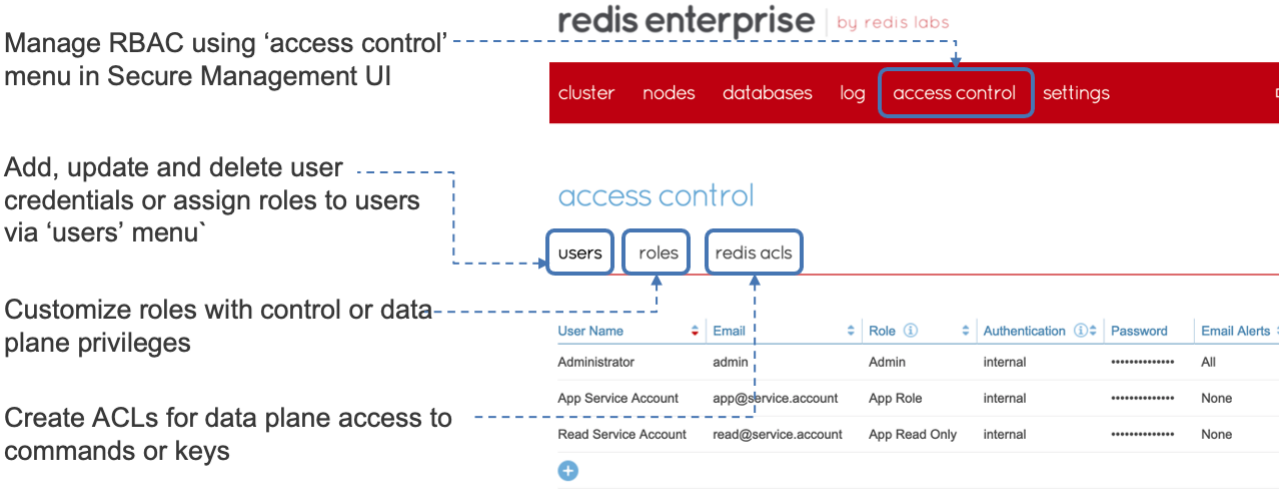
Roles

- Control plane permissions
- Data plane permissions
- Can be applied to one, many or all databases

ACLs

- Grant access to specific commands
- Grant access to specific keys
- Redis Open Source format

You have a centralized interface which provides a clear management layer to view and control who has access to the system and databases.

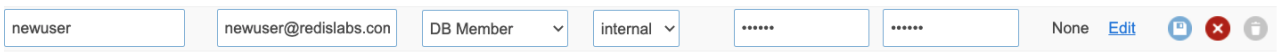


User Management

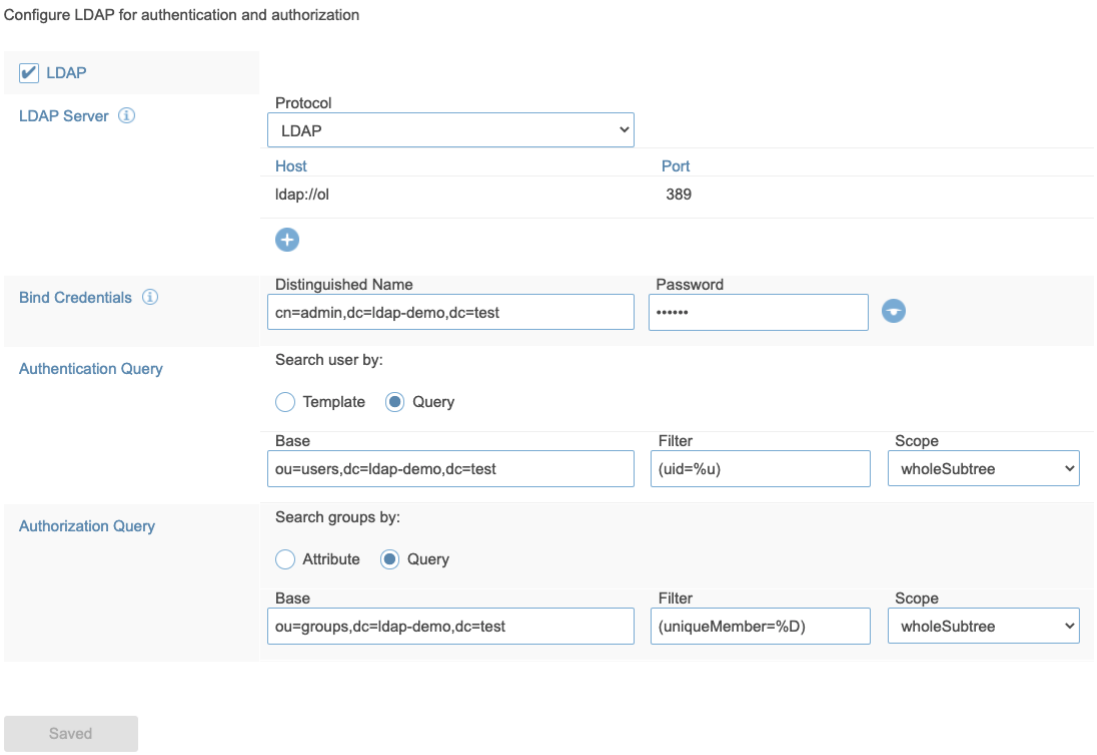
You can add new users using the Redis Enterprise management layer:

- Grant users access to the cluster or databases
- Create service accounts for applications
- Each user can have a role assigned to it
- Internal password authentication
- Identities can be tied to account management systems through LDAP
- Email alerts can be configured

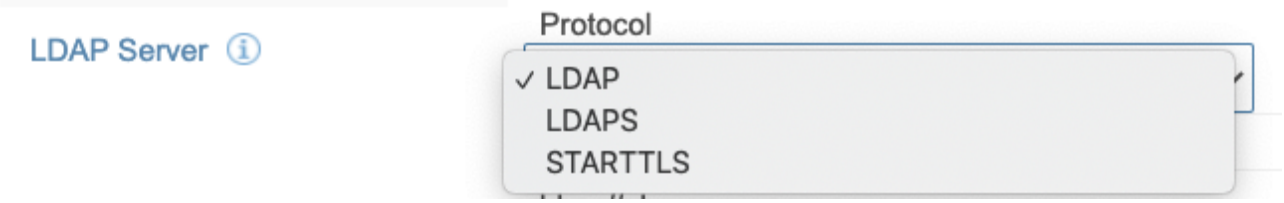
Users created in the Redis Enterprise with a password through the admin console or REST API.



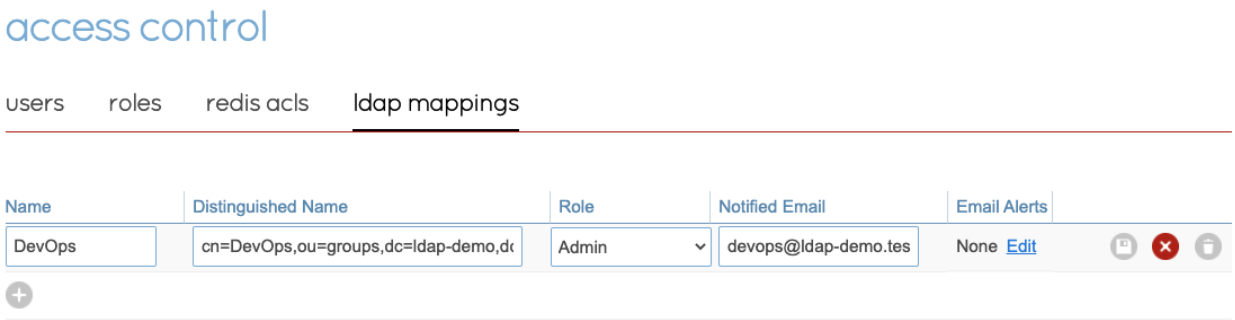
Additionally, you can also grant users access using your organization's central identity management system via LDAP. Your organization's LDAP endpoint/s and auth/z settings can be configured.



There is support for LDAPS and STARTTLS protocols as well.



Once the LDAP connection is configured an LDAP group can be mapped to a Redis Enterprise role.



In the above example, the DevOps group is mapped to the ‘Admin’ role in Redis Enterprise so any user that is part of that DevOps group will gain admin level permissions automatically.

Once the mappings are set up the actual user credentials and users can all be managed centrally in your organization.

Roles

When creating users or mapping your LDAP groups, you can assign a role to a user. Roles can be applied to multiple users to provide the appropriate level of permissions. This allows a user to be connected to control plane permissions for cluster management or viewing and/or Redis ACLs for database permissions.

Control Plane Roles

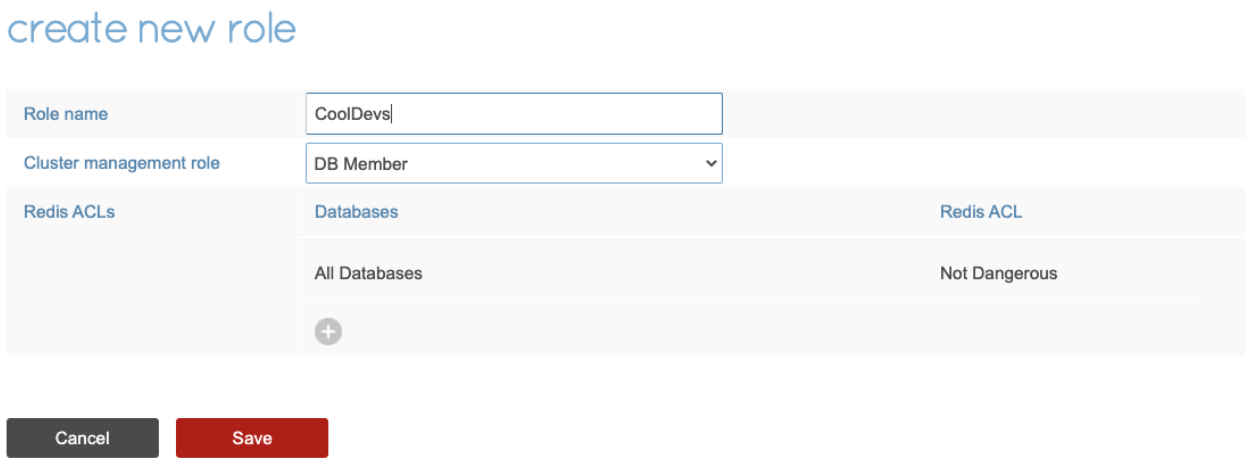
Redis Enterprise connects resources (VM or physical instance) into a Redis Enterprise Cluster. Once a cluster is created Redis databases can be created across those resources. This Redis Enterprise Cluster administration is what we would call the control plane.

You can see from the control plane roles and their descriptions below that besides the Admin role the access is split between ‘Cluster’ and the ‘DB’ functions.

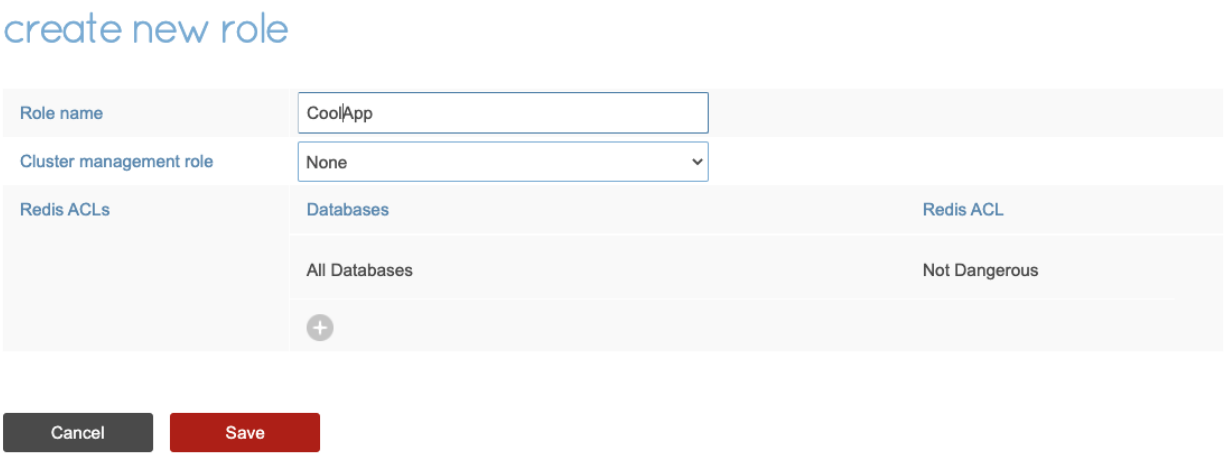
RBAC allows you to grant access to DevOPs users to administer the cluster and Dev users to administer their particular databases by just assigning the correct role.

You can also create new roles. These roles can permit access to both the data and administration level access.

For example, suppose you want to grant a group of developers access to manage and view Redis databases configurations and stats on the Redis Enterprise cluster. In this case, you could create a role that included both the control plane role of ‘DB Member’ along with database access.



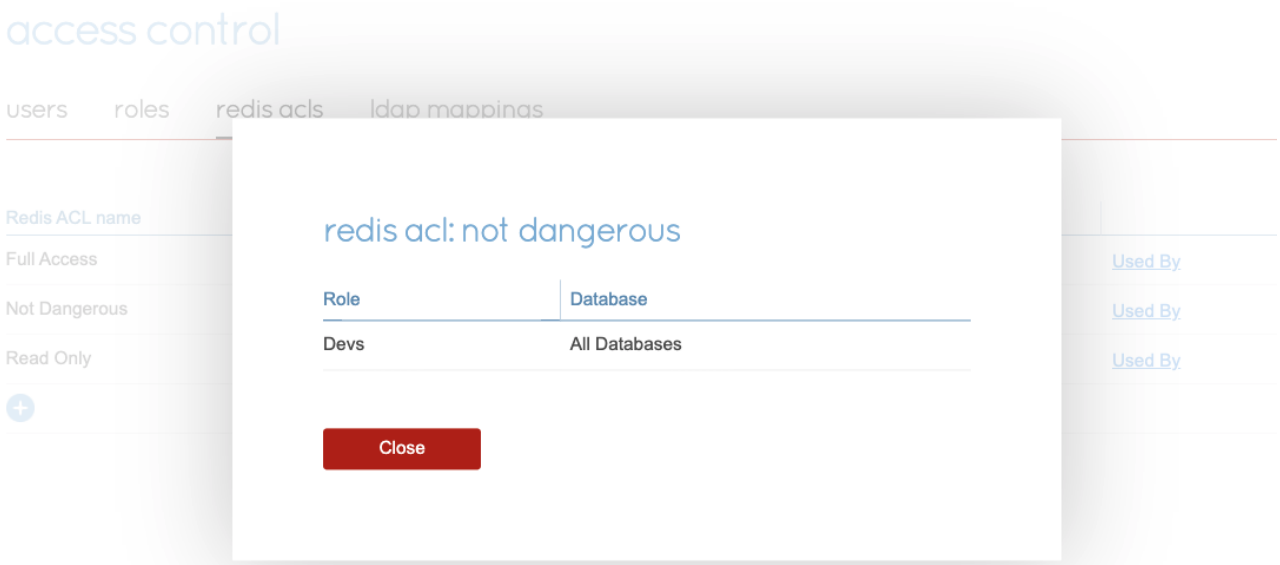
You can also create application-specific access roles. These roles don't need to provide any cluster management capabilities; rather, they provide only the database access needed by the application.



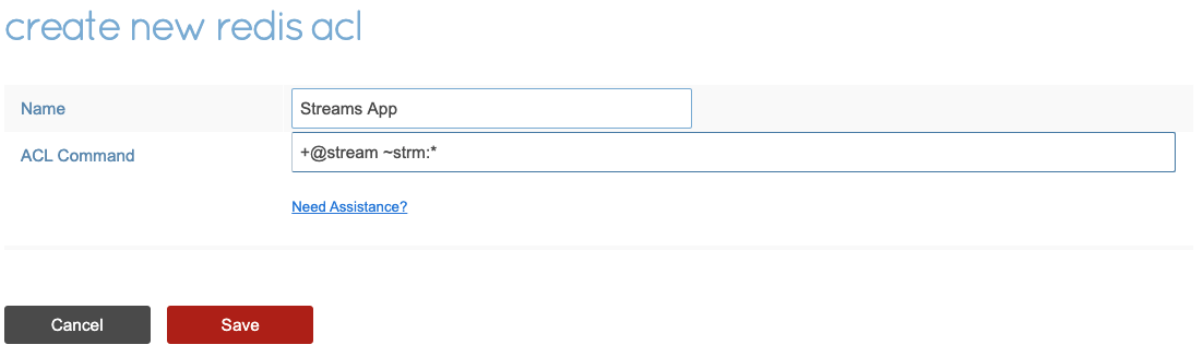
Roles can be database specific. For example, you could update this role for ‘CoolApp’ to only have access to ‘DB1’ and not all of the databases.[5] [6]

ACLs

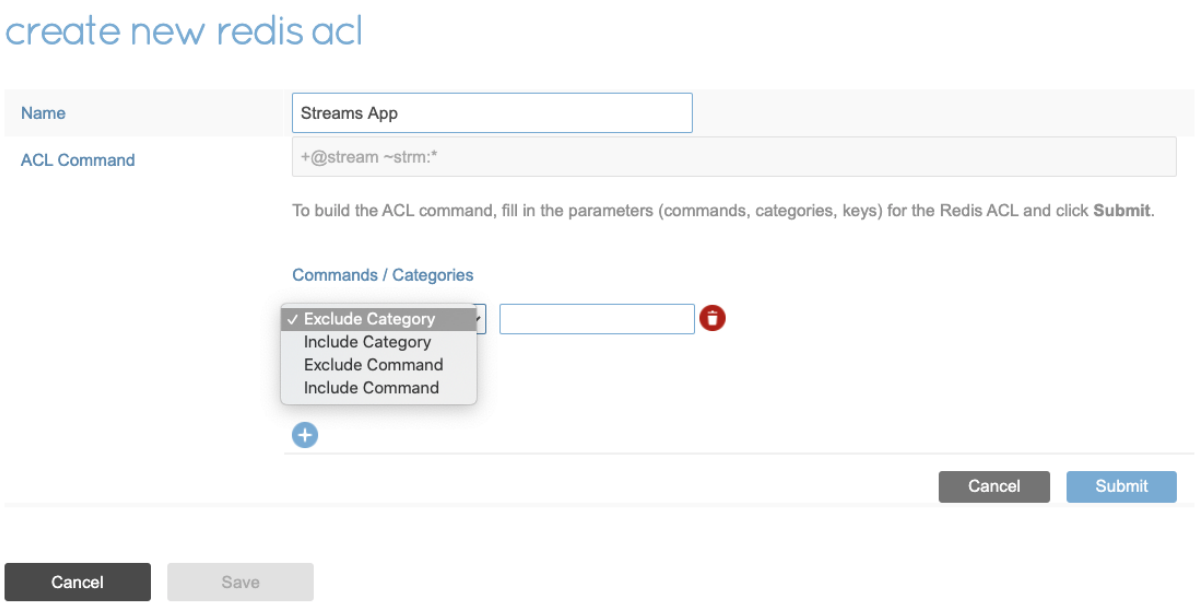
We began to touch on roles for database access in the last section which showed how a Redis ACL can be applied to a role. Redis ACLs can be added and maintained separately from the user which significantly reduces management overhead and provides view into your database access exposure. On each ACL there is a ‘Used By’ view to see which Role and databases it has been applied.



There are a few ACLs predefined to get started with, but you can also create custom ACLs.



If you have gone through the previous section on Redis ACLs, then the syntax should look familiar. Redis Enterprise provides an ACL builder utility, in case you're not familiar with the ACL syntax. .



Once an ACL is created, you can assign it to a role and database/s. The great thing about having the ACL decoupled from the user (and even the role) is that you can update the ACL and immediately apply the changes to all associated users and roles as your requirements change.

2.9 Exercise: RBAC in Redis Enterprise

Set Up

Clone this repo if you have not already: https://github.com/redislabs-training/exercises-scaling-redis

Change into the security-rbac directory.

Requirements

- docker
- docker-compose
- internet connection

Starting Environment

```
docker-compose up -d
```

After the `docker-compose -d` command has successfully started up the Redis Enterprise docker container we will need to bootstrap Redis Enterprise. This involves creating a single-node cluster. Note that in Redis Enterprise, a cluster node equates to an instance resource on which Redis Enterprise is installed and not just where a Redis server process is running.

There is a `setup.sh` script provided which will get things started so you can focus on the RBAC exercises.

Run from this directory:
`./setup.sh`

Once that completes you should be able to load the Redis Enterprise admin console in a browser:

```
https://localhost:8443/
```

Redis Enterprise creates self-signed certs at installation time so that a secure connection is always created. Since these certificates are not CA signed certificates, you will need to allow your browser to accept them.

In the Firefox browser this consists of just clicking through the Advanced button and finding the button to accept and continue. If you are using a Chrome browser, you can bypass this warning by typing `'thisisunsafe'`, and it will automatically continue through to the site.

Once you bypass the browser warning, you should see a login screen where you can use the following credentials to log in:

username: `admin@redis.com`

password: `redis123`

Once successfully logged in, you should see the Redis Enterprise admin console and be taken to create a database immediately.

Leave all the defaults and click 'Next', which will take you to a screen where you can name the database, keep the existing defaults and click 'Activate' to create the database.

Now you should be ready for the RBAC exercises.

Connect to the database

Redis Enterprise supports multi-tenancy (a single cluster can host separately managed databases) so a unique port and endpoint are created per database.

1. Find the database in the 'databases' menu and click on it
2. View the 'Endpoint' ... grab the port value in particular
3. Connect to redis-cli in the docker container on that port by running:

```
docker-compose exec redis_enterprise_rbac redis-cli -p <port-from-endpoint>
```

If you are successfully connected, you should be able to run the INFO command.

Update Default User

1. Find the database in the databases menu and click the Edit button for the database
2. Add a password for the default user
3. Verify using the redis-cli from the first step and after connecting use the AUTH command with the new password
4. Disconnect from the database

Create a simple command ACL and update it

1. Navigate to the 'access control' menu
2. View the default roles and redis acls

3. Add a new Redis ACL where you restrict commands to GEO by removing all commands and adding the geo category for all keys
4. Create a new role that has no control-plane access
5. Grant this role access to the database you created (or all databases) and add the new Redis ACL you just created
6. Now go to the users menu and create at least one new user with this role
7. Verify your changes were correct:
8. `redis-cli` into the db as we did before
9. `AUTH <new-user> <new-user-pwd>`
10. run a `GEO command`:

```
GEOADD Sicily 13.361389 38.115556 "Palermo"
15.087269 37.502669 "Catania"
```

d. run a string command `SET fa la` and you should get a NOPERM error.

1. Now go back to the admin console and find the 'redis acls' menu again. Append the ACL you just created with '+@string' and Update it.
2. Go back to the redis-cli prompt and re-run the string command 'set fa la' and it should succeed
3. Disconnect from your database

Create a key restrictive ACL

1. Navigate back to 'redis acls' menu inside of 'access control' tab
2. Add a new Redis ACL that allows for access to all non-dangerous commands and keys starting with 'public:'
3. Apply this ACL to your database or all databases
4. Now navigate to 'roles' and add a new role that utilizes the ACL you just created
5. Now add this role to a new user
6. Verify your changes were correct:
7. `redis-cli` into the db as we did before
8. `AUTH <new-user> <new-user-pwd>`
9. create new key/s prefixed with `'public:'` like `'public:example'` without a NOPERM error
10. create new key/s not prefixed with `'public:'` and get a NOPERM error

11. You can experiment with updating the key restriction with additional key matches or command inclusion/exclusions and then verifying them via redis-cli
12. Disconnect from your database

Create new user with control plane access

1. In the Redis Enterprise admin console navigate to the users are again under the access control menu
2. Use the plus button to create a new user
3. Fill in all the values with some fake/test values (you will need to remember the email and password) except for Role choose 'DB Viewer'
4. Now in another browser or in an 'incognito' type browser tab open `https://localhost:8443` and login with this new user you just created
5. You should be presented with the databases section only. If you select the database you created notice that you can view the metrics page. If you click on configuration the edit capability is disabled.
6. If you select the access control nav menu you only have the ability to change your own password.

Stopping Environment

```
docker-compose down
```

Unit 3

Course: Running Redis at Scale
Unit 3: Persistence/Durability

3.1 Persistence options in Redis

If a server that stores data in RAM is restarted, all data is lost. To prevent such data loss, there needs to be some mechanism for persisting the data to disk; Redis provides two of them: snapshotting and an append-only file (AOF). You can configure your Redis instances to use either of the two, or a combination of both of them.

Snapshotting

When a snapshot is created, the entire point-in-time view of the dataset is written to persistent storage in a compact `.rdb` file. You can set up recurring backups, for example every 1, 12, 24 hours and use these backups to easily restore different versions of the data set in case of disasters. You can also use these snapshots to create a clone of the server, or simply leave them in place for a future restart.

Creating a .rdb file requires a lot of disk I/O. If performed in the main Redis process, this would reduce the server's performance. That's why this work is done by a forked child process. But even forking can be time-consuming if the dataset is large. This may result in decreased performance or in Redis failing to serve clients for a few milliseconds or even up to a second for very large datasets. Understanding this should help you decide whether this solution makes sense for your requirements.
[1] [2]

You can configure the name and location of the `.rdb` file with the `dbfilename` and `dir` configuration directives, either through the `redis.conf` file, or through the `redis-cli` (as explained in chapter 1.3):

```
dbfilename my_backup_file.rdb
dir ./rdbs
```

And of course you can configure how often you want to create a snapshot. Here's an excerpt from the redis.conf file showing the default values:

```
# Unless specified otherwise, by default Redis
will save the DB:
#   * After 3600 seconds (an hour) if at least 1
key changed
#   * After 300 seconds (5 minutes) if at least 100
keys changed
#   * After 60 seconds if at least 10000 keys
changed
#
# You can set these explicitly by uncommenting the
three following lines.
#
# save 3600 1
# save 300 100
# save 60 10000
```

As an example, this configuration will make Redis automatically dump the dataset to disk every 60 seconds if at least 1000 keys changed:

```
save 60 1000
```

While snapshotting is a great strategy for the use cases explained above, it leaves a huge possibility for data loss. You can configure snapshots to run every few minutes, or after X writes against the database, but if the server crashes you will lose all the writes since the last snapshot has been done.

In many use cases, that kind of data loss can be acceptable, but in many others it is absolutely not. For all of those other use cases Redis offers the AOF persistence option.

AOF

AOF, or append-only file works by copying every incoming write command to disk as it happens. These commands can then be replayed at server startup, to reconstruct the original dataset. Commands are logged using the same format as the Redis protocol itself, in an append-only fashion.

Using AOF Redis is much more durable than snapshotting: you can choose to force flushing all modified buffer cache pages for the AOF file to disk by an explicit call of the system `fsync` function. Depending on your durability requirements (or how much data can you afford to lose), you can choose which `fsync` policy is the best for your use case:

- `fsync every write` - The safest policy: The write is acknowledged to the client only after it has been written to the AOF file and flushed to disk. Since in this approach we are writing to disk synchronously, we can expect a much higher latency than usual.
- `fsync every second`: The default policy. Fsync is performed asynchronously, in a background thread, so write performance is still high. Choose this option if you need high performance and can afford to lose up to one second worth of writes.
- **no fsync**: In this case Redis will log the command to the file descriptor, but will not force the OS to flush the data to disk. If the OS crashes we can lose a few seconds of data (Normally Linux will flush data every 30 seconds with this configuration, but it's up to the kernel's exact tuning.).

The relevant configuration directives for AOF are:

```
# AOF Enabled
appendonly no

# The name of the append only file (default:
"appendonly.aof")
appendfilename "my_aof_file.aof"

# fsync policy (options are everysec, no and
always. Default is everysec)
appendfsync everysec
```

AOF contains a log of all the operations that modified the database in a format that's easy to understand and parse. When the file gets too big, Redis can automatically rewrite it in the background, compacting it in a way that only the latest state of the data is preserved.

3.2 Exercise - Saving a Snapshot

As we learned in the previous chapter, Redis will save a snapshot of your database every hour if at least one key has changed, every five minutes if at least 100 keys have changed, or every 60 seconds if at least 10000 keys have changed.

Let's update this to a simplified hypothetical scenario where we want to save a snapshot if three keys have been modified in 20 seconds.

Step 1

Create a directory named 3.2 and in it prepare a `redis.conf` file.

```
$ mkdir 3.2
$ cd 3.2
$ vim redis.conf
```

The `redis.conf` file should specify a filename that will be used for the rdb file and a directive that will trigger the creation of a snapshot if 3 keys have been modified in 20 seconds, as described above.

```
dbfilename my_backup_file.rdb
save 20 3
```

Step 3

While inside of the 3.2 directory start a Redis server, passing it the redis.conf configuration file you just created.

```
$ redis-server ./redis.conf
```

In a separate terminal tab use the redis-cli to create three random keys, one after the other. For example:

```
> SET a 1
> SET b 2
> SET c 3
```

Run the `ls` command in the 3.2 directory to list all its contents. What changed?

Step 4

Now we're ready to take our persistence a level higher and set up an AOF file. Modify your redis.conf file so that the server will log every new write command and force writing it to disk.
Be careful! We have a running server and we want this configuration to be applied without restarting it.

```
> CONFIG SET appendonly yes
> CONFIG SET appendfsync always
```

In order for these settings to be persisted to the redis.conf file we need to save them:

```
> CONFIG REWRITE
```

Step 5

Create a few random keys through `redis-cli`. Check the contents of the directory 3.2 again. What changed?

Step 6

As a final step, restart the Redis server process (you can press Ctrl+C in the terminal to stop the process and re-run it again). If you run the `SCAN 0` command you will see that all the keys you created are still in the database, even though we restarted the process.

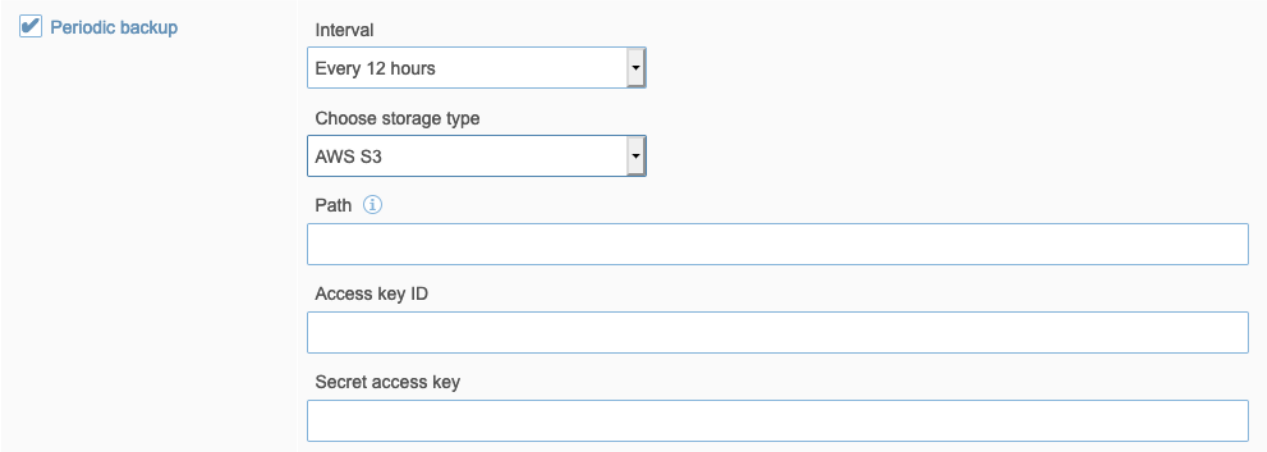
3.3 Redis Enterprise Backup and Restore

As you have seen, Redis can be configured to take snapshots periodically. As you start to scale out your database into multiple shards, this means each shard will have its own snapshot of the data, typically spread across multiple physical or virtual machines. Redis Enterprise also supports AOF and snapshots for persistence but also adds the ability to take a backup of the whole database, aggregating the different snapshots, and storing them in a remote location.

Scheduled Backups

The backup feature can be enabled to run at scheduled intervals for each of your databases. This can be accessed in the admin console. Once the periodic backup is selected for a database, there

are options for interval, storage type, and other details to connect with the storage type.

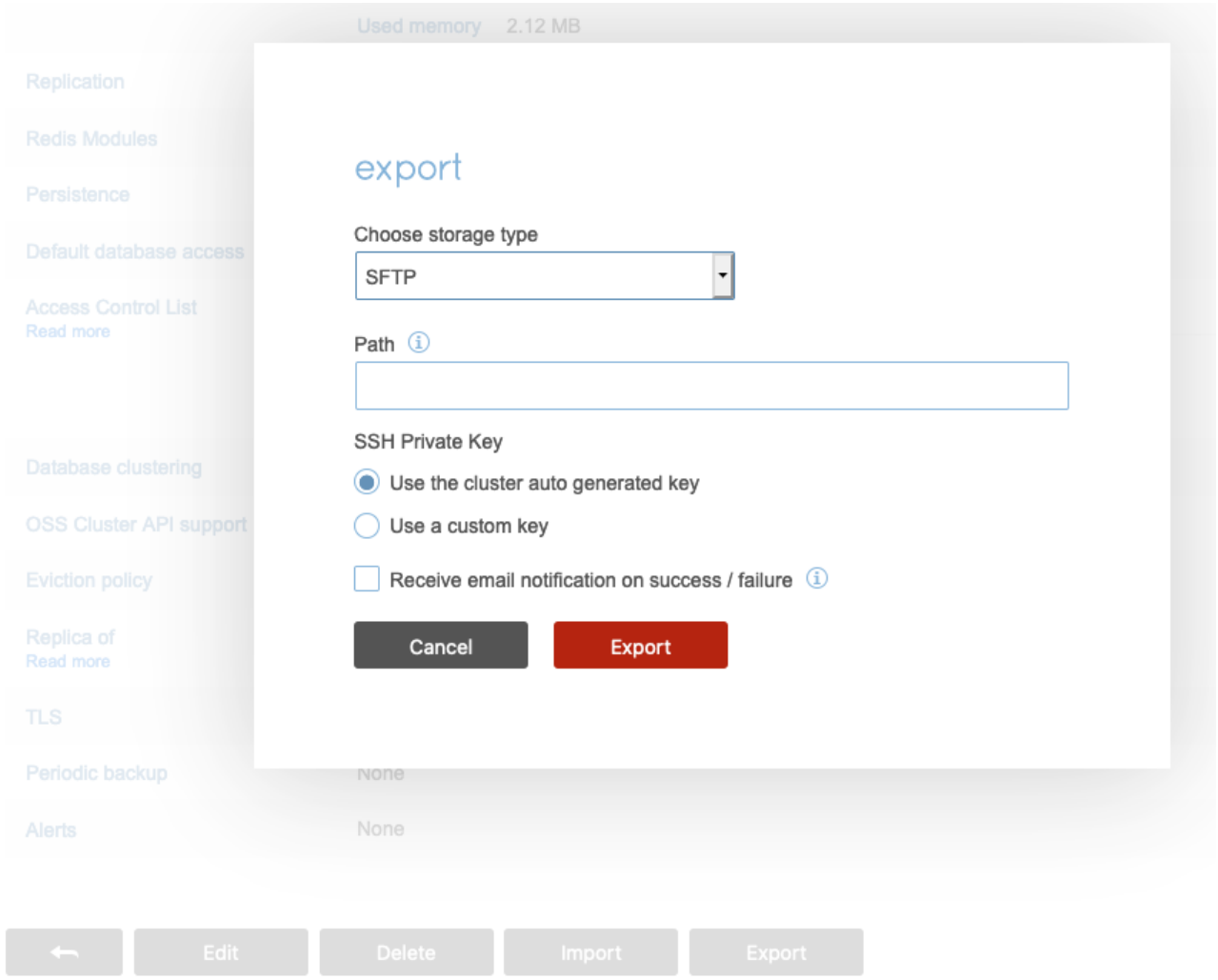


The storage types supported are:

- AWS S3
- Azure Blob Storage
- FTP / SFTP
- Google Cloud Storage
- Mount Point
- OpenStack Swift

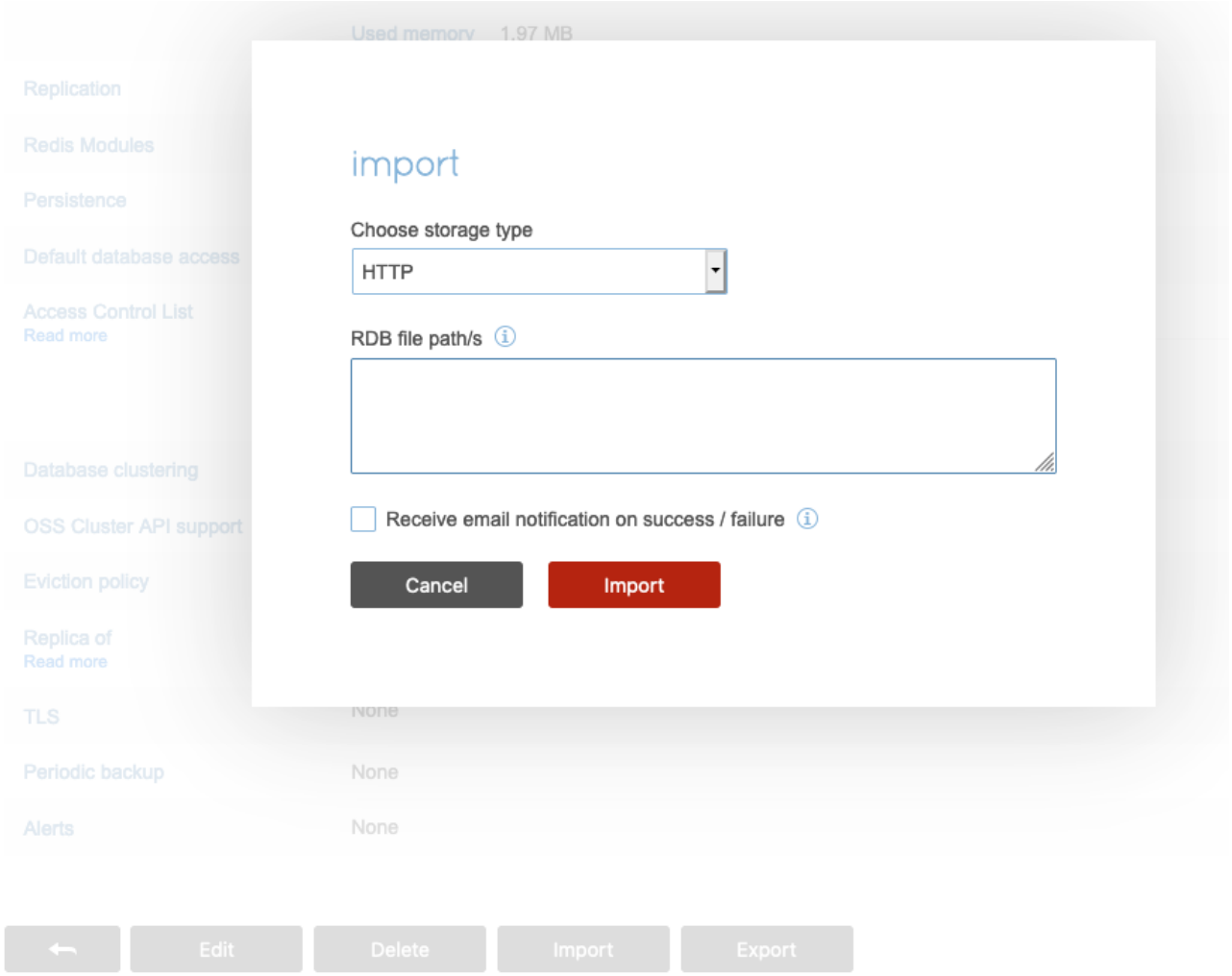
Ad-hoc Backups

There is also the ability to trigger a backup on demand using the same storage types. This can be something that you run right before an event like patching the systems in your Redis Enterprise cluster. The database configuration area of the admin console has an 'Export' button which starts this backup process.



Restore

The Redis Enterprise backups can be restored to a Redis Enterprise database using the 'Import' button in the database configuration. You can pull the backup from all the same storage types and an additional 'HTTP' type.



Note: this will reset the existing database before it does the import.

Verification

It is always good practice to test your backup and restore procedure. This will give you a good idea of what you can expect from this process. Some questions to consider include:

- How long does it take to run the backup (especially when going over the network)?
- How long did it take to restore?
- How large is the backup?
- When restoring is all the expected data recovered?

Unit 4: High Availability

Course: Running Redis at Scale

4.1 Introduction to High Availability

High availability is a computing concept describing systems that guarantee a high level of uptime, designed to be fault-tolerant, highly dependable, operating continuously without intervention and without a single point of failure.

What does this mean for Redis specifically? Well, it means that if your primary Redis server fails, a backup will kick in, and you, as a user, will see little to no disruption in the service.

There are two components needed for this to be possible: replication and automatic failover.

Replication is the continuous copying of data from a primary database to a backup, or a replica database. The two databases are usually located on different physical servers, so that we can have a functional copy of our data in case we lose the server where our primary database sits.
But having a backup of our data is not enough for high availability. We also have to have a mechanism that will automatically kick in and redirect all requests towards the replica.
This mechanism is called automatic failover.

In the rest of this chapter we'll see how Redis handles replication and which automatic failover solutions it offers. Let's dig in.

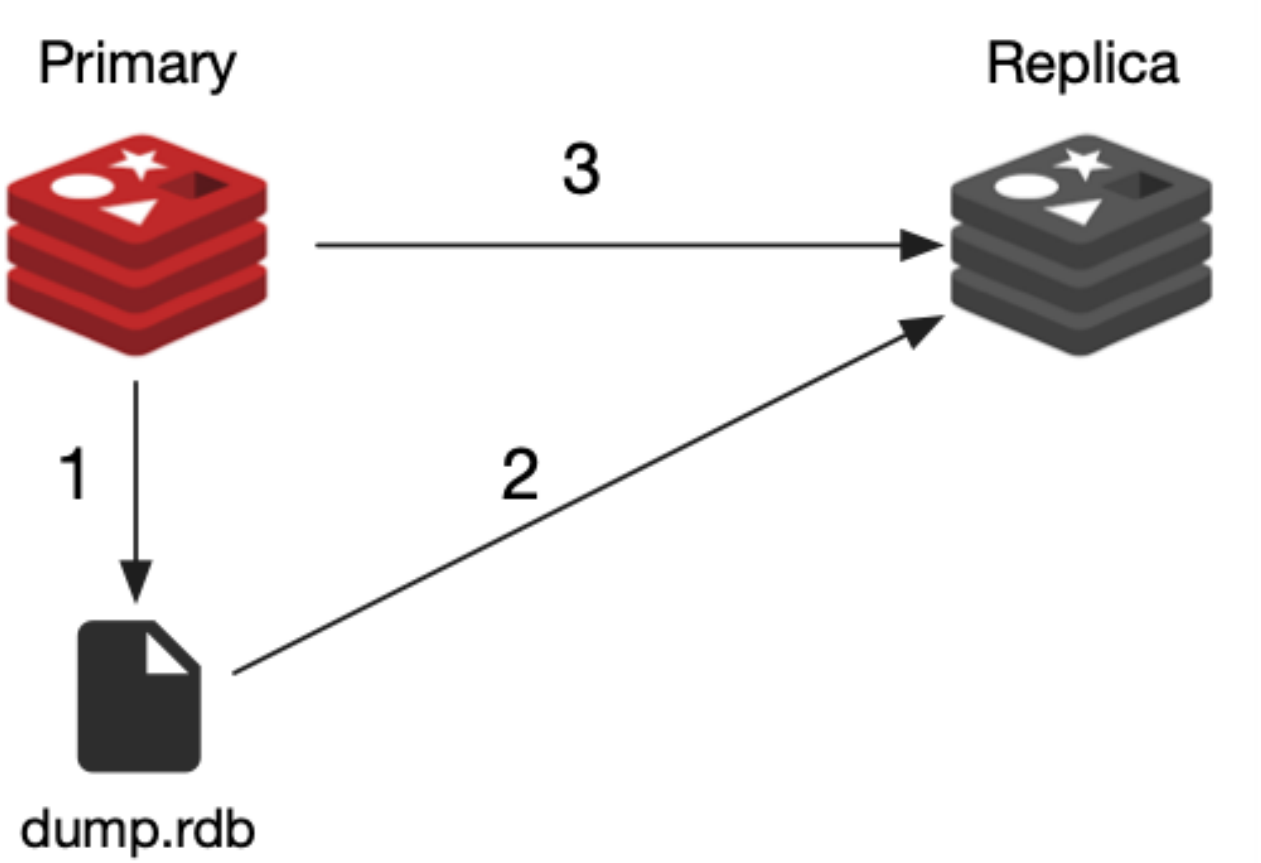
4.2 Basic Replication

Replication in Redis follows a simple primary-replica model where the replication happens in one direction - from the primary to one or multiple replicas. Data is only written to the primary instances and replicas are kept in sync so that they're exact copies of the primaries.

Creating a replica is simple: you need to instantiate a Redis server instance with the configuration directive `replicaof` set to the address and port of the primary instance:

```
replicaof 192.168.1.1 6379
```

Once the replica instance is up and running, the replica will try to sync with the primary. In order to transfer all of its data as efficiently as possible, the primary instance will produce a compacted version of the data in a snapshot (.rdb) file and send it to the replica.



The replica will read the snapshot file and load all of its data into memory, which will bring it to the same state the primary instance had at the moment of creating the rdb file. When the loading stage is done, the primary instance will send the backlog of all write commands that have been executed against it in those few seconds between the snapshot creation and the moment it was loaded in the replica.
Finally, the primary instance will start sending to the replica a live stream of all commands it receives from clients.

If at some point we want to connect our replica to a different primary instance, we can do that by issuing the `REPLICAOF` command (it's worth noting that if there is any data on a replica instance when replication starts, it will get deleted).

```
replicaof 192.168.1.2 6379
```

Or we can configure it to not be a replica of anyone anymore:

```
replicaof no one
```

Data safety with replication

By default, replication is asynchronous, which means that if you send a write command to Redis (1) you will receive your acknowledged response first (2), and only then will the command be replicated to the replica (3).

If the primary goes down after acknowledging a write but before the write can be replicated, then you might have data loss.

To avoid this, you can use the **`WAIT`** command. This command blocks the current client until all of the previous write commands are successfully transferred and acknowledged by at least the specified number of replicas.

For example, if we send the command **`WAIT 2 0`**, the client will block (will not return a response to the client) until all of the previous write commands issued on that connection have been replicated to at least 2 replicas.

The second argument (0) will instruct the server to block indefinitely, but we could set it to a number (in milliseconds) so that it times out after a while and returns the number of replicas that successfully acknowledged the data.

Using replicas to scale reads

By default, replicas are configured to be read-only, which means you can configure your clients to read from them, but you cannot **write** data to them. SInce the replication direction is one-way only (primary to replica), this is understandable; if we write a key named foo to the replica it will get overwritten as soon as some other client creates a key with the same name on the primary.

But having read-only replicas can be useful too, and not just for high availability, but also for increasing the performance in terms of throughput we can get out of our system. Namely, you can configure your clients to read data from the replicas as well as primary instances; assuming you have two replicas, now you can have three Redis servers processing your requests. When reading from a single primary, we only have one server processing our commands, one at a time. With one primary and two replicas

serving our reads, we get three commands being processed at the same time, increasing our throughput by a factor of three

4.3 Exercise - Enabling basic replication

Step 1

First let's create and configure the primary instance. We'll start with a few configuration changes in its primary.conf configuration file.

```
$ touch primary.conf # Create the configuration
file
```

Now open the primary.conf file with your favourite text editor and set the following configuration directives:

```
# Create a strong password here
requirepass a_strong_password

# Enable AOF file persistence
appendonly yes

# Choose a name for the AOF file
appendfilename "primary.aof"
```

Finally, let's start the primary instance:

```
$ redis-server ./primary.conf
```

Step 2

Next, let's prepare the configuration file for the replica:

```
$ touch replica.conf
```

Let’s add some settings to the file we just created:

```
# Port on which the replica should run
port 6380

# Address of the primary instance
replicaof 127.0.0.1 6379

# AUTH password of the primary instance
masterauth a_strong_password

# AUTH password for the replica instance
requirepass a_strong_password
```

And let’s start the replica:

```
$ redis-server ./replica.conf
```

Step 3

Open two terminal tabs and use them to start connections to the primary and replica instances:

```
# Tab 1 (primary)
$ redis-cli

# Tab 2 (replica)
$ redis-cli -p 6380
```

Authenticate on both tabs by running the command AUTH followed by your password:

```
AUTH a_strong_password
```

On the second (replica) tab run the **MONITOR** command which will allow you to see every command executed against that instance.

Go back to the first (primary) tab and execute any write command, for example

```
> SET foo bar
```

In the second tab you should see that the command was already sent to the replica:

```
1617230062.389077 [0 127.0.0.1:6379] "SELECT" "0"
1617230062.389092 [0 127.0.0.1:6379] "set" "foo"
"bar"
```

Step 4

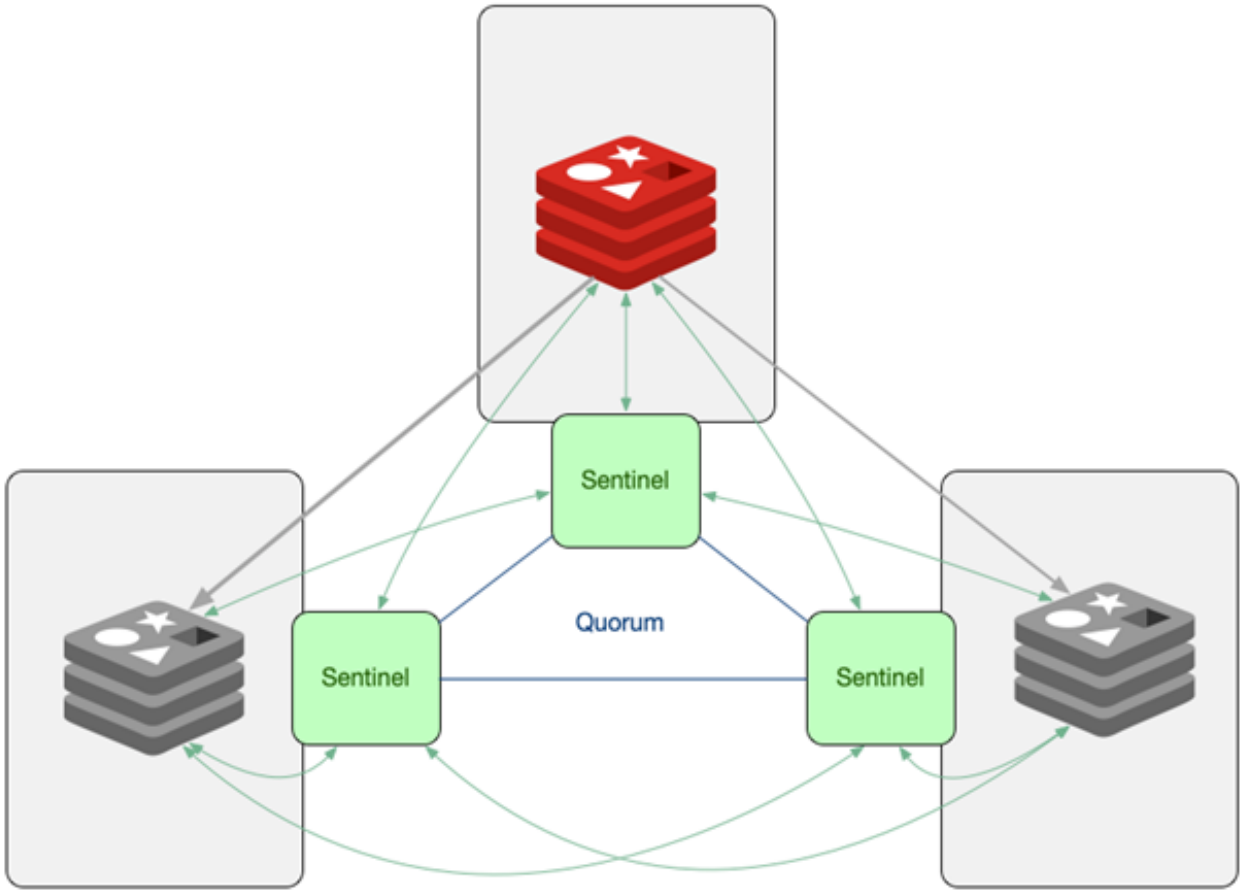
Keep the instances running, or at least their configuration files around. We'll need them for the next exercise.

4.4 Understanding Sentinel

In the beginning of this unit, we learned that we can't have high availability without replication and automatic failover. We covered replication in the previous two chapters, and now we'll explain Sentinel - a tool that provides the automatic failover.

Redis Sentinel is a distributed system consisting of multiple Redis instances started in sentinel mode. We call these instances **Sentinels**.
The group of Sentinels monitors a primary Redis instance and its replicas. If the sentinels detect that the primary instance has failed, the sentinel processes will look for the replica that has the latest data and will promote that replica to be the new primary. This way, the clients talking to the database will be able to reconnect to the

new primary and continue functioning as usual, with minimal disruption to the users.



Deciding that a primary instance is down
In order for the Sentinels to be able to decide that a primary instance is down we need to have enough Sentinels agree that the server is unreachable from their point of view.
Having a number of Sentinels agreeing that they need to take an action is called **reaching a quorum**. If the Sentinels can't reach quorum, they cannot decide that the primary has failed.
The exact number of Sentinels needed for quorum is configurable.

Triggering a failover
Once the Sentinels have decided that a primary instance is down, they need to elect and authorize a leader (a Sentinel instance) that will do the failover. A leader can only be chosen if the majority of the Sentinels agree on it.

In the final step, the leader will reconfigure the chosen replica to become a primary by sending the command `REPLICAOF NO ONE` and it will reconfigure the other replicas to follow the newly promoted primary.

Sentinel and client libraries
If you have a system that uses Sentinel for high availability, then you need to have a client that supports Sentinel. Not all libraries have this feature, but most of the popular ones do, so make sure you add it to your list of requirements when choosing your library.

4.5 Exercise - Sentinel hands-on

Step 1

If you still have the primary and replica instances we set up in the previous exercise (4.3) - great! We'll reuse them to create our Sentinel setup. If not - well, just go back to the instructions and go through them again.

When done, you will have a primary Redis instance with one replica.

Step 2

To initialise a Redis Sentinel, you need to provide a configuration file, so let's go ahead and create one:

```
$ touch sentinel1.conf
```

Open the file and paste in the following settings:

```
port 5000
```

```
sentinel monitor myprimary 127.0.0.1 6379 2
sentinel down-after-milliseconds myprimary 5000
sentinel failover-timeout myprimary 60000
sentinel auth-pass myprimary a_strong_password
```

port - The port on which the Sentinel should run

sentinel monitor - monitor the Primary on a specific IP address and port. Having the address of the Primary the Sentinels will be able to discover all the replicas on their own. The last argument on this line is the number of Sentinels needed for quorum. In our example - the number is 2.

sentinel down-after-milliseconds - how many milliseconds should an instance be unreachable so that it's considered down

sentinel failover-timeout - if a Sentinel voted another Sentinel for the failover of a given master, it will wait this many milliseconds to try to failover the same master again.

sentinel auth-pass - In order for Sentinels to connect to Redis server instances when they are configured with requirepass, the Sentinel configuration must include the sentinel auth-pass directive.

Step 3

Make 2 more copies of this file - sentinel2.conf and sentinel3.conf and edit them so that the PORT configuration is set to 5001 and 5002, respectively.

Step 4

Let's initialise the three Sentinels in three different terminal tabs:

```
# Tab 1
$ redis-server ./sentinel1.conf --sentinel
```

```
# Tab 2
$ redis-server ./sentinel2.conf --sentinel

# Tab3
$ redis-server ./sentinel3.conf --sentinel
```

Step 5

If you connected to one of the Sentinels now you'd be able to run many new commands that would give an error if run on a Redis instance. For example:

```
# Provides information about the Primary
SENTINEL master myprimary

# Will give you information about the replicas
connected to the Primary
SENTINEL replicas myprimary

# Will provide information on the other Sentinels
SENTINEL sentinels myprimary

# Provides the IP address of the current Primary
SENTINEL get-master-addr-by-name myprimary
```

Step 6

If we killed the primary Redis instance now by pressing Ctrl+C or by running the `redis-cli -p 6379 DEBUG sleep 30` command, we'll be able to observe in the Sentinels' logs that the failover process will start in about 5 seconds. If you run the command that returns the IP address of the Primary again you will see that the replica has been promoted to a Primary:

```
> SENTINEL get-master-addr-by-name myprimary
1) "127.0.0.1"
2) "6380"
```

4.6 High Availability in Redis Enterprise

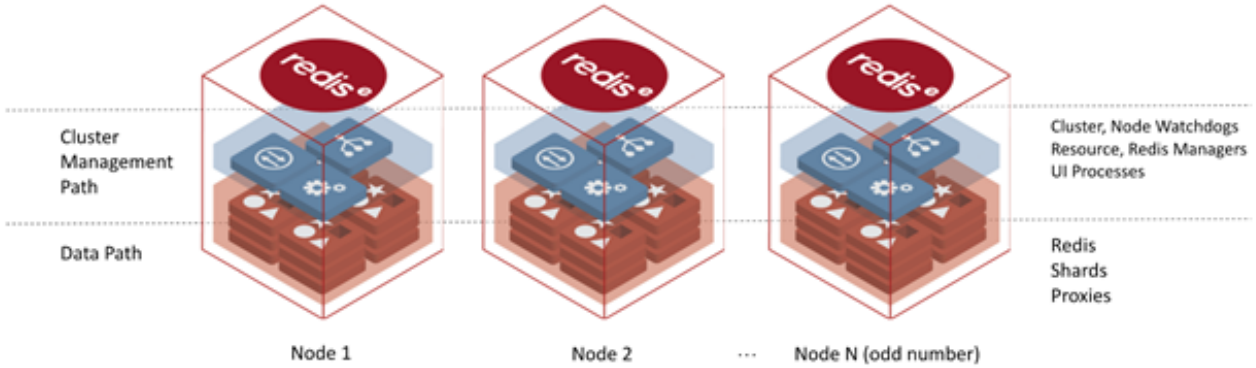
In a production environment, you need to ensure the availability of your data which means that you can't just look at the Redis server processes themselves; you will need to detect machine failures, network hiccups, zones dying or even whole data centers or regional failures.

Since Redis Enterprise was developed as a DBaaS in the cloud from the beginning, it was designed with features to maintain availability through all these failure scenarios.

Disconnected or Failed Resources
You are probably familiar with the concept of quorum and Redis (with replication and sentinel), but it is important to note that Redis Enterprise maintains quorum differently.

A Redis Enterprise node is not just a Redis server process but a resource (such as a physical/virtual machine, container or cloud instance). In this context, maintaining quorum means having the minimum number of resources online and connected.

To maintain quorum, Redis Enterprise clusters should always have an odd number of nodes, with three nodes being the minimum.

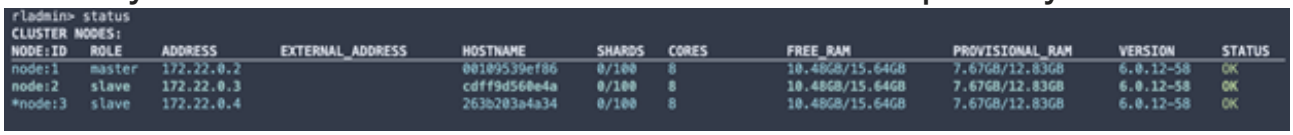


The communication between nodes is monitored through watchdog processes, so the loss of a node, whether through failure or networking split, is detected and reacted to immediately by the cluster manager process.

Redis Enterprise is deployed in a shared-nothing architecture so the remaining nodes can continue to function with everything they need, even in the event of a failure. One of the node's cluster manager processes is elected as the 'primary' for the cluster and orchestrates cluster operations.

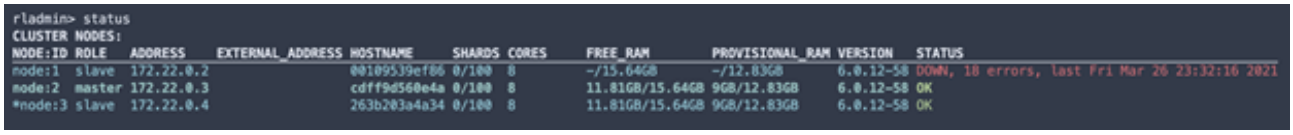
Here is an example. Using the rladmin CLI utility (which ships with Redis Enterprise), we can view the state of the cluster and make cluster configuration changes (in this case we will just view the status).

Healthy state: 3-node cluster and `node:1` is the primary node:



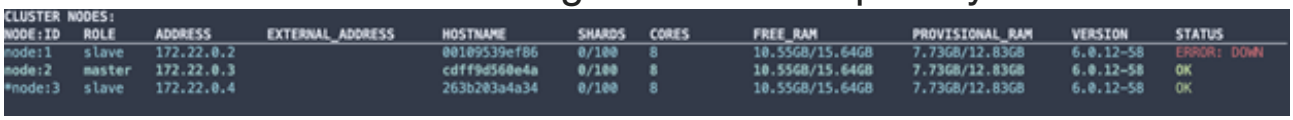
```
rladmin> status
CLUSTER NODES:
NODE:ID   ROLE    ADDRESS      EXTERNAL_ADDRESS  HOSTNAME      SHARDS  CORES  FREE_RAM          PROVISIONAL_RAM   VERSION    STATUS
node:1    master  172.22.0.2                    00109539ef86  0/100   8      10.48GB/15.64GB   7.67GB/12.83GB    6.0.12-58  OK
node:2    slave   172.22.0.3                    cdff9d560e4a  0/100   8      10.48GB/15.64GB   7.67GB/12.83GB    6.0.12-58  OK
*node:3   slave   172.22.0.4                    263b203a4a34  0/100   8      10.48GB/15.64GB   7.67GB/12.83GB    6.0.12-58  OK
```

`node:1` is disconnected from the cluster and in a `'DOWN'` state so `node:2` is elected as the primary node:



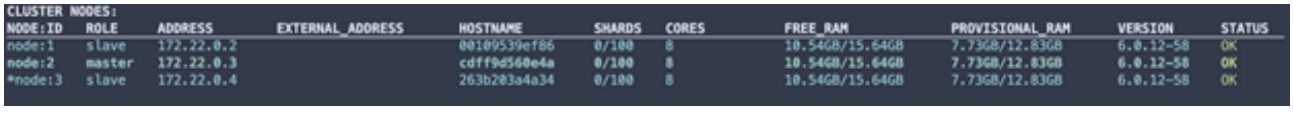
```
rladmin> status
CLUSTER NODES:
NODE:ID  ROLE    ADDRESS      EXTERNAL_ADDRESS  HOSTNAME      SHARDS  CORES  FREE_RAM          PROVISIONAL_RAM   VERSION    STATUS
node:1   slave   172.22.0.2                    00109539ef86  0/100   8      -/15.64GB         -/12.83GB         6.0.12-58  DOWN, 18 errors, last Fri Mar 26 23:32:16 2021
node:2   master  172.22.0.3                    cdff9d560e4a  0/100   8      11.81GB/15.64GB   9GB/12.83GB       6.0.12-58  OK
*node:3  slave   172.22.0.4                    263b203a4a34  0/100   8      11.81GB/15.64GB   9GB/12.83GB       6.0.12-58  OK
```

`node:1` returns but the configuration is not quite synced:



```
CLUSTER NODES:
NODE:ID   ROLE    ADDRESS      EXTERNAL_ADDRESS  HOSTNAME      SHARDS  CORES  FREE_RAM          PROVISIONAL_RAM   VERSION    STATUS
node:1    slave   172.22.0.2                    00109539ef86  0/100   8      10.55GB/15.64GB   7.73GB/12.83GB    6.0.12-58  ERROR: DOWN
node:2    master  172.22.0.3                    cdff9d560e4a  0/100   8      10.55GB/15.64GB   7.73GB/12.83GB    6.0.12-58  OK
*node:3   slave   172.22.0.4                    263b203a4a34  0/100   8      10.55GB/15.64GB   7.73GB/12.83GB    6.0.12-58  OK
```

`node:1` fully restored to the cluster but is no longer the primary node of the cluster:



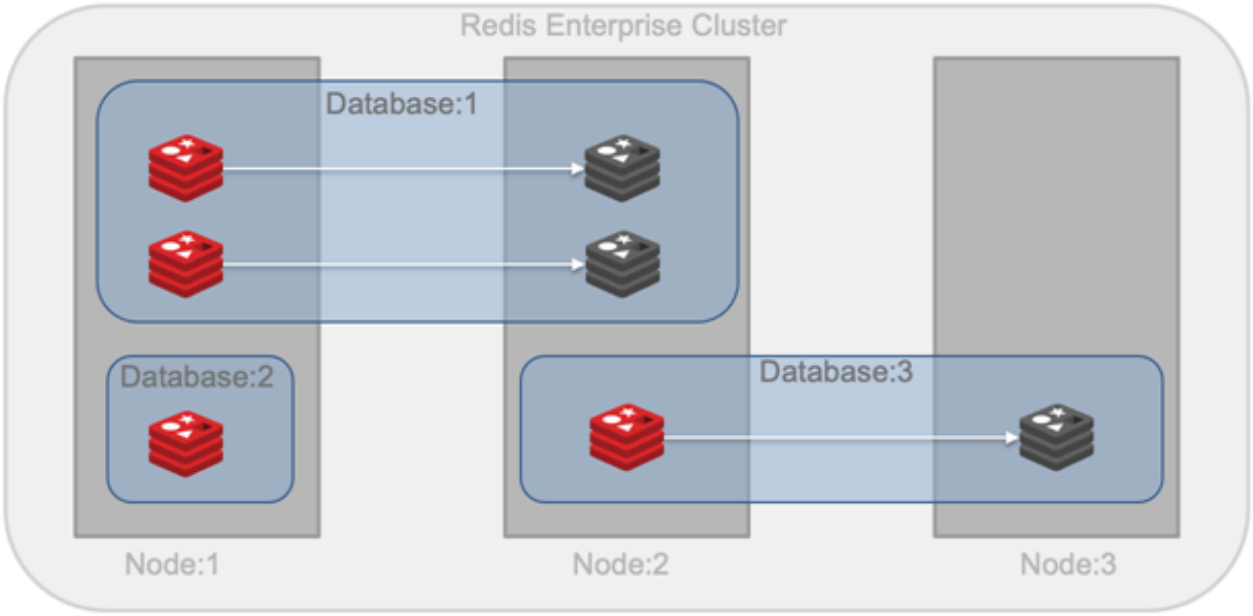
```
CLUSTER NODES:
NODE:ID   ROLE    ADDRESS      EXTERNAL_ADDRESS  HOSTNAME      SHARDS  CORES  FREE_RAM          PROVISIONAL_RAM   VERSION    STATUS
node:1    slave   172.22.0.2                    00109539ef86  0/100   8      10.54GB/15.64GB   7.73GB/12.83GB    6.0.12-58  OK
node:2    master  172.22.0.3                    cdff9d560e4a  0/100   8      10.54GB/15.64GB   7.73GB/12.83GB    6.0.12-58  OK
*node:3   slave   172.22.0.4                    263b203a4a34  0/100   8      10.54GB/15.64GB   7.73GB/12.83GB    6.0.12-58  OK
```

When there is a loss or failure of the majority of nodes, quorum is lost, and the full functionality will be limited until the nodes are recovered. Each Redis Enterprise node persists its cluster management configurations in the case that the nodes need to be re-created to be recovered.

Maintaining the availability of enough resources within the cluster allows for another set of high availability features to be enabled.

Redis Process Management and Shard Failures

Multi-Tenancy:
The quorum of nodes design allows Redis Enterprise to maintain separate Redis databases as separate entities with their own resource and availability requirements.



In the diagram above:

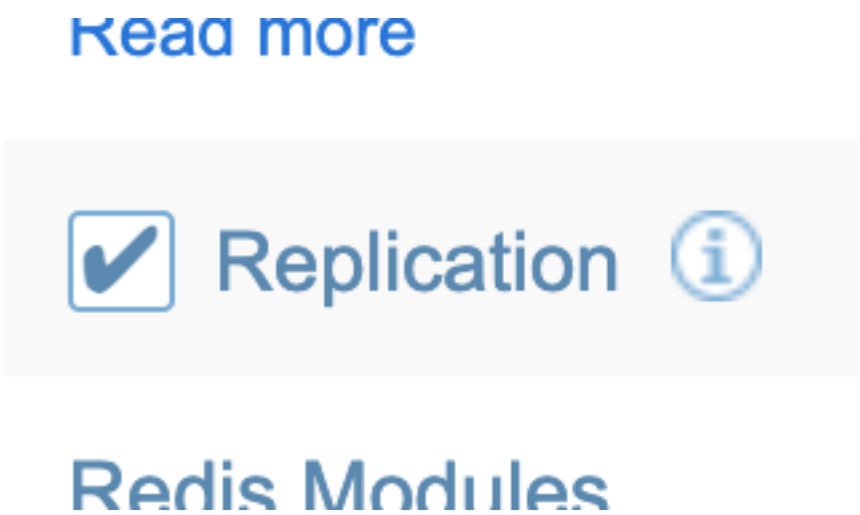
- Database 1 is clustered into two shards with replication enabled

- Database 2 is not clustered with no replication
- Database 3 is not clustered with replication

The three databases have different configurations and are managed separately according to those configurations by the Redis Enterprise cluster manager.

Replication Management:

Redis Enterprise provides fully automated replication. Replication can be enabled in the admin console by simply checking a box in the database configuration (or through the management layer REST API):

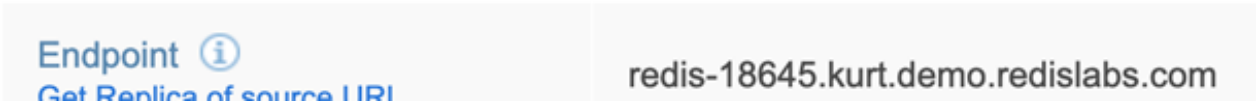


Once replication is enabled, the cluster manager will determine a place to host the replica shard/s. It will never host the replica shard on the same node as a primary shard.

Handling Shard Failures: Preventing Loss of Access
Redis Enterprise has several processes that work together to handle the failure of a shard: DNS, the proxy, and watchdogs.

Typically, a DNS nameserver record is configured to point to the cluster so it can handle the domain and all sub domains. Each node runs an authoritative dns server which is configured by the cluster manager. This allows for specific sub domain endpoints to be created for each database.

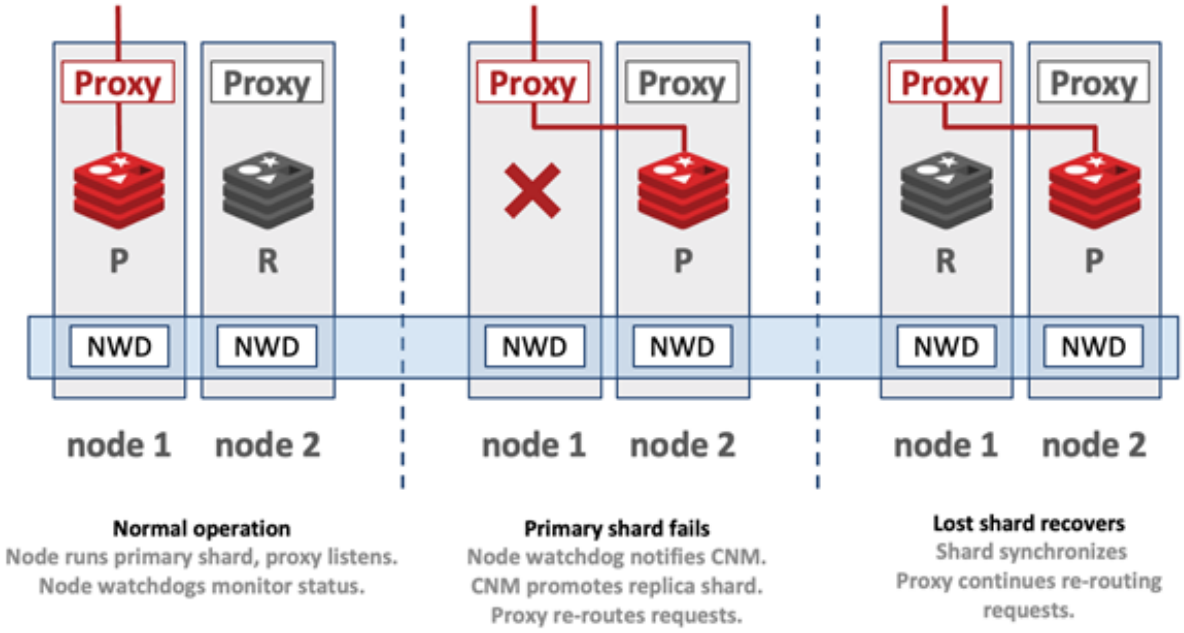
Shown from the database configuration details in the admin console:



Then, depending on the database configuration, one or multiple proxy services will be enabled across the cluster. When the client performs the DNS lookup for the database endpoint,, the request will be routed to one of the node's DNS servers. The response will include the IPs of the proxies that are connected to that database.

While the proxy provides significant performance gains through multiplexing and pipelining, it also allows for failover to happen transparently to the client.

Each node runs a watchdog process, which will detect the failure of a shard and communicate that to the cluster manager. This detection is critical for triggering the failover from a primary to a replica to avoid downtime.



The watchdog will also detect a proxy failure and fail over the proxy endpoint to another node. This ensures that clients can still connect to the hosted Redis databases.

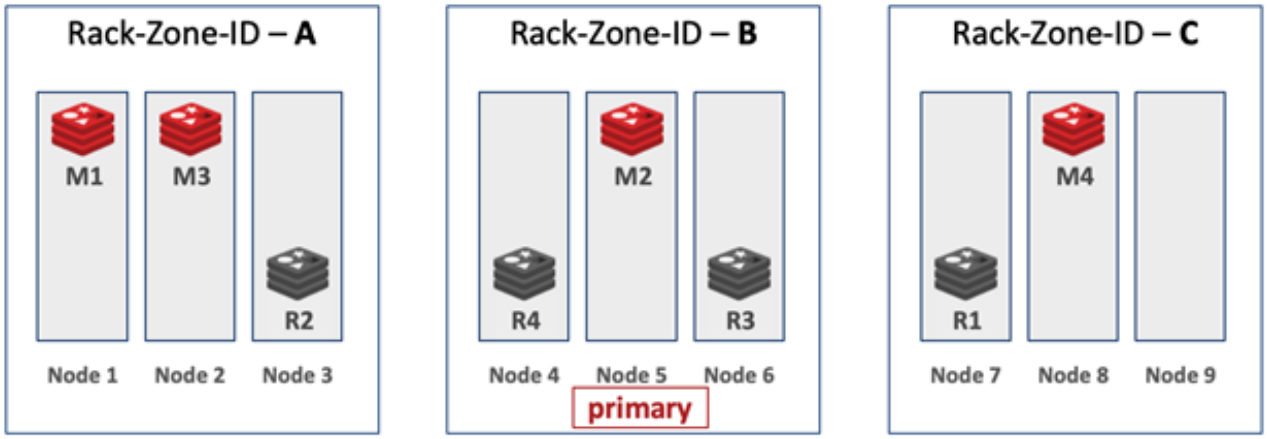
From a user perspective, you just enable replication, and then all of the hard work of placing replicas and handling failover is done for you. This ease of operation is one of the main reasons why we designed Redis Enterprise.

Availability Zone or Rack Failures

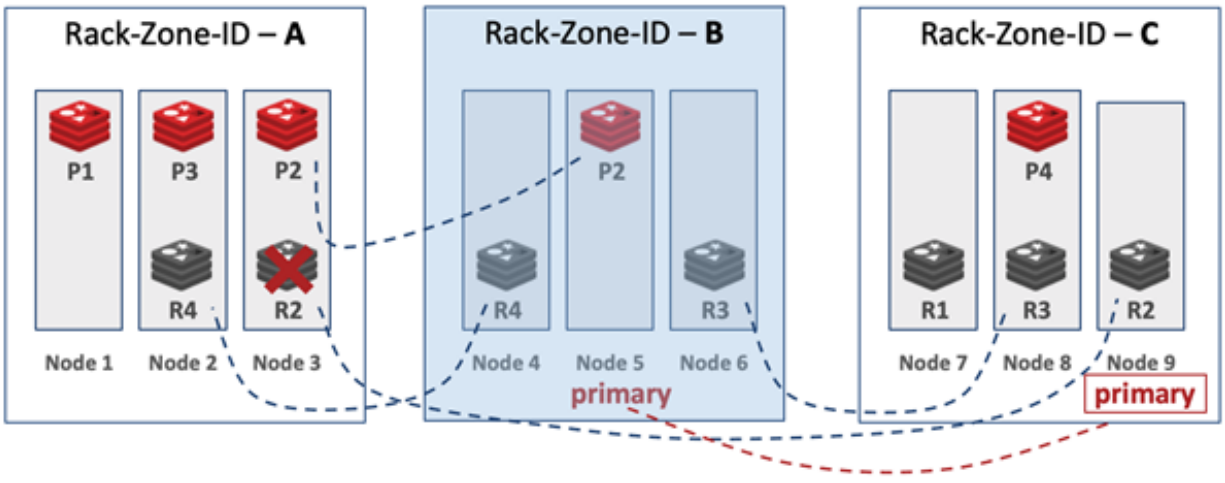
Another failure case you will want to guard against is the loss of a whole availability zone or rack. It's common to deploy resources across racks and zones, but ensuring that the deployment properly utilizes these zones to provide high availability is non-trivial.

Redis Enterprise has built-in logic to take advantage of different racks/zones. This is known as rack-zone awareness.

Shards are dispersed across rack-zones so that, if a rack-zone goes down, replica shards get promoted in other rack-zones, and the database stays online.



If a rack (or zone) goes down (say B), primary shards in that rack (P2) get shards (S2) promoted. Replica shards in that rack (S4, S3) start on new nodes.



Also notice from the diagrams above that the primary cluster manager node will also fail over, and a new cluster manager will be elected to maintain cluster operations.

You configure a node's zone when the node joins the cluster. After that, the cluster manager uses the zone in its shard placement decisions.

Regional (or Data Center) Failures

Disaster recovery requirements have become common and sometimes difficult to manage. Preventing a zone failure is one thing, but how can you survive a complete regional failure? What if a data center becomes unavailable? Redis Enterprise can enable a replicated database across multiple regions. There are two main options.

Active-Passive Replication

Redis Enterprise supports a passive cluster as a cold standby with a backup database. The database can be configured to be a `'replicaOf'` a database in another cluster. The replication will only take place from the source to the replica.

This is easily configured during the creation of the database in the passive cluster in the database configuration management. The

primary endpoint can be configured along with whether TLS should be used or not.

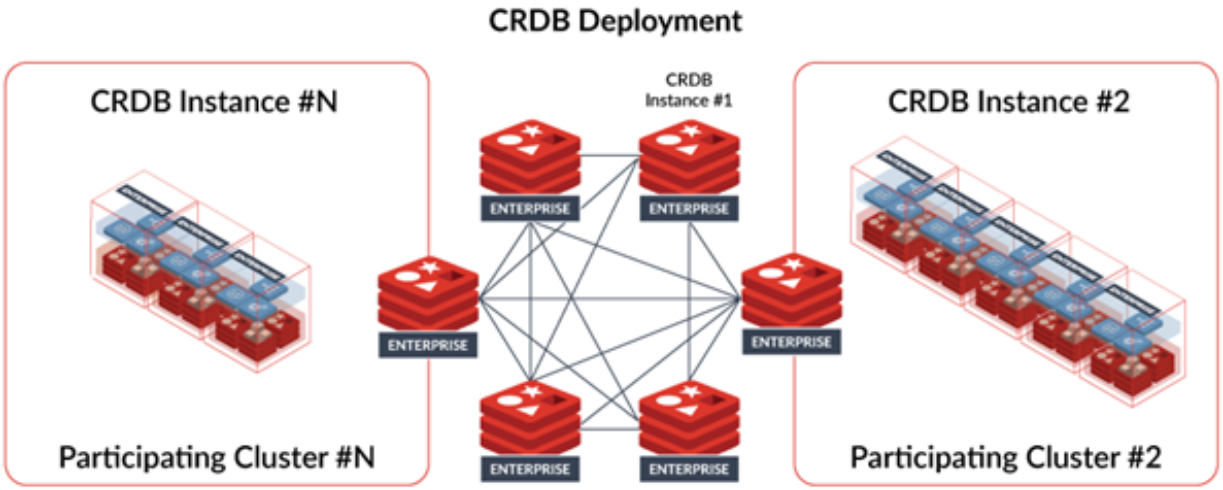
While this seems like a simple solution, some care should be taken when planning your disaster recovery scenarios. Since the replication is one-way, going back to the original source database cannot be automatic. A solution needs to be thought through to replicate any changes that could have occured on the replica database to the original source.

Active-Active Replication

A more robust solution is Redis Enterprise's Active-Active. In an Active-Active deployment, a single database is replicated multi-directionally across geographically-distributed clusters. This means that you can actively write to and read from all participating databases, and all databases will remain in sync.

This is achieved by the Redis Enterprise implementation of CRDTs (conflict-free replicated data types).

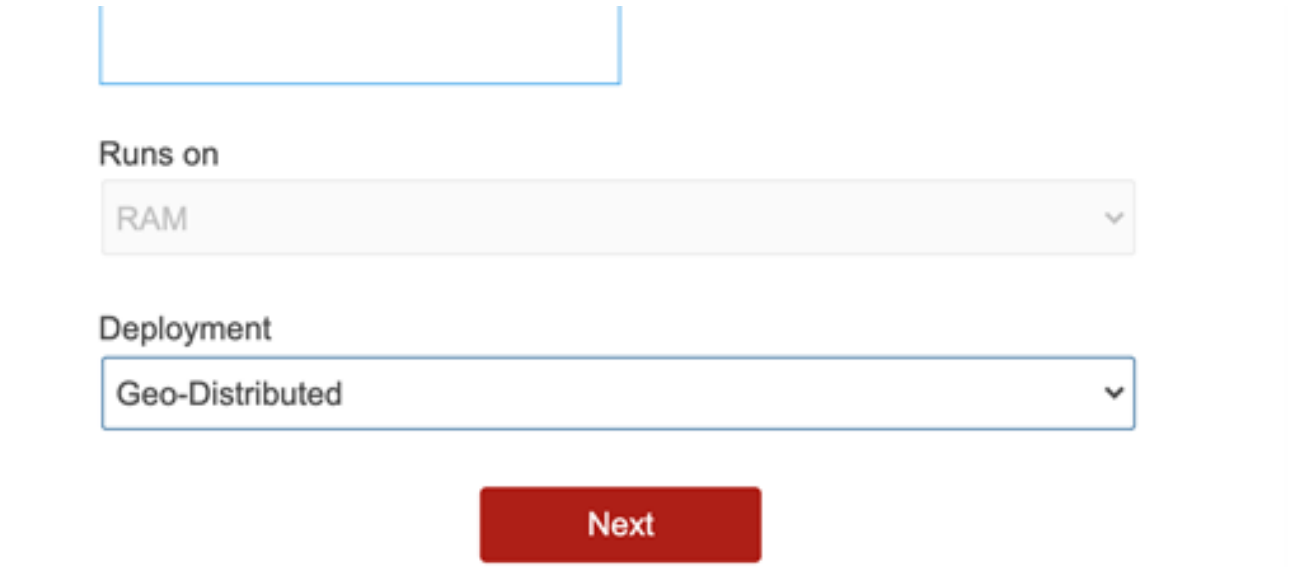
So we sometimes call these databases “conflict-free replicated databases” or “CRDBs.” These databases offer local latency on read and write operations while enabling seamless conflict resolution (“conflict-free”).



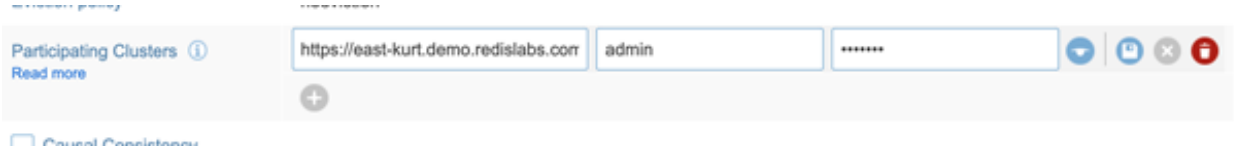
An application can read/write from any instance or easily switch between instances without planning or migration. Even if the majority of geo-replicated regions in a CRDB (for example 3 out of 5) are down, the remaining geo-replicated regions will remain uninterrupted and can continue to handle read and write operations.

Running an Active-Active database requires a Redis Enterprise cluster to be set up in each of the regions you wish to replicate to. Once you have your clusters deployed, you can create your Active-Active databases using the admin console.

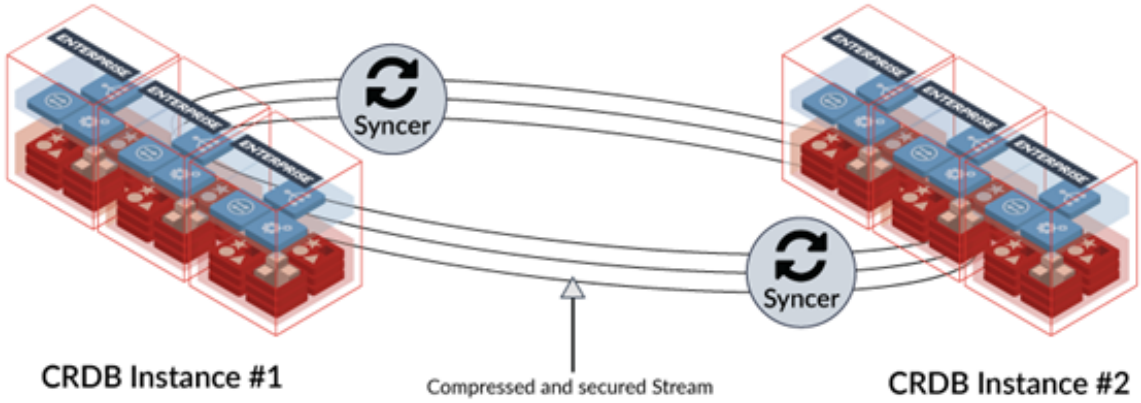
First, indicate that you wish to create a Geo-Distributed database during the database creation wizard:



Then in the database configuration, you can add the participating clusters:



Once an Active-Active database is being replicated across regions there are processes running to keep the different instances synchronized.



if there is a network partition or failure in one of the instances, the remaining continue to read and write. In the case of regional failures, clients that can't connect to a local CRDB instance can be diverted to other data centers that point to one of the available CRDB instances.

In rare occasions, a CRDB instance may experience full data loss and need replication from scratch. There is a reconciliation mechanism that involves all of the relevant CRDB instances. Once reconciled, the recovering instance can simply do a full sync from any other replica.

4.7 Exercise - Enabling High Availability

Clone this repo if you have not already: https://github.com/redislabs-training/exercises-scaling-redis

Change into the 'ha-management' directory.

Requirements

- docker
- docker-compose
- internet connection

Starting Environment

```
docker-compose up -d
```

Getting started

After the **`docker-compose -d`** command has successfully started up the Redis Enterprise docker container we check to make sure that internally Redis Enterprise is ready to start taking requests. Run from this directory:

```
./verify-readiness.sh
```

Once that completes with a successful message you should be able to proceed to:

https://localhost:18443/

Redis Enterprise creates self-signed certs in the beginning so a secure connection is always created. Since they are not CA signed certificates you will need to allow your browser to accept them.

Depending on your browser and version you may have an 'advanced' option that allows you to accept and continue/proceed. Some Chrome browser versions will not have this option, but you can bypass this warning by just typing `'thisisunsafe'` and it will automatically continue through to the site.

Once you bypass the browser warning you should see the initial setup screen.

Create cluster on node 1
You will see a large red 'Setup' button. Click that to begin setting up the Redis Enterprise cluster.

The node configuration page will show some initial defaults including local file paths to use, IPs, etc. You can take these defaults. Please note the IP of this first node as you will need it in subsequent steps.

Make sure the 'Create new cluster' is selected and then enter a 'Cluster name' value: `cluster.local` and click next (leaving the other checkboxes empty).

NOTE: In this local setup using docker we will not be able to fully use the DNS capabilities of Redis Enterprise.

The next screen will prompt you for cluster authentication by asking you to enter a license key. Just click next here and a trial license will automatically be used.

Now you will need to set admin credentials. These will be used in subsequent steps. Here is an example of something that will work:

```
Email: admin@redis.com
Password: redis123
```

After credentials have been set you will be taken immediately to a screen to create a database. We do not need to do that yet.

Join cluster on node 2
Now let's move to node 2 and join this node to the Redis Enterprise cluster. Enter this address in the browser address bar:

https://localhost:28443

This will again require you to bypass the self-signed certificate warning in the browser. Please refer to the instructions above to bypass.

You will again be prompted with the large red setup button, click that to proceed. On the node configuration page you will now select 'Join cluster' in the cluster configuration area. Enter the IP of the first node `(172.22.0.11)` and the credentials you created, then click Next.

Normally after a cluster is created on a node or a node joins a cluster Redis Enterprise will update the self-signed certs with the cluster domain name configured. In this exercise environment where we keep re-using localhost and don't have unique IPs that we can enter into the host browser it may cause issues. You might need to reload your browser a few times which may lead to accepting the self-signed certs again and another login.

Once you get past all of this you will be taken to a screen to create a database. As you have noticed this is where Redis Enterprise will redirect you if no database exists yet.

Let's leave the browser at this place and open up a terminal.

Join cluster on node 3 using rladmin
You could go through the same steps that we did for node 2 to join node 3 to the cluster, but let's take the opportunity to see that there are other options. Note: there are three ways to create a cluster... admin console in the browser, `rladmin` CLI and the administration REST API.

The `rladmin` CLI utility is fairly easy to access in this dockerized exercise environment so let's use that. First connect to node 3 shell:
`docker-compose exec re3 bash`

This should land you on a prompt like this (with a diff container ID):
`redislabs@cc13e2721cd9:/opt$`

Now run `rladmin` to get the interactive mode of the utility. At the `rladmin>` prompt you can enter the tab key twice to view a list of available commands. Since this node is not currently set up there is really only one command `cluster` (besides exit or help).

Enter `cluster` followed by the tab key twice to view a list of available options....now add `join` to the command. You can enter the tab key twice to view a list of available options:

```
rladmin> cluster join
accept_servers        cnm_https_port        flash_en
abled nodes
password              replace_node
addr                  ephemeral_path        flash_pa
th
override_rack_id      persistent_path       required
_version
ccs_persistent_path   external_addr         json_fil
e
override_repair       rack_id               username
```

We will not use all of these, but you might remember the admin console node configuration screen during setup and notice some similar fields. Since our setup is simple and we can take most of the defaults let's just provide a minimal command to join this node to the cluster:

```
rladmin> cluster join nodes 172.22.0.11 username
admin@redis.com password redis123
Joining cluster... ok
rladmin>
```

You will need to replace the above username and password with the credentials that you created when you created the cluster.
With the cluster joined you should now have more command options available to you. Enter the tab key twice to view a list of available commands which should include different cluster management commands that can be used to view cluster configurations or even update them.
View the cluster status by entering:

```
rladmin> status
```

You should have a display with different sections:

- `CLUSTER NODES`
- `DATABASES`
- `ENDPOINTS`
- `SHARDS`

Currently you should only have data in the `CLUSTER NODES` section since you have not added a database yet. Speaking of...

Create a database

Return to the browser window you left before. The session has likely timed out which will require you to login again. After a successful login you should now be presented with the prompt to create a new database. Select 'redis database,' 'Runs on ram' and 'Single Region' and click 'Next.'

You should now have a screen to enter the database details. Let's keep it simple to start with by entering a name and checking the 'Replication' box then click the 'Activate' button. Redis Enterprise will create a single sharded database with replication using it's state machine to stand up necessary resources and connect them together.

That's it, you just created a database with high availability!

Replication details
Return to the terminal and run `rladmin status` again and you should see additional sections containing data about your database. Each database has an endpoint and shards associated with it. If you look at the `SHARDS` section in detail you should see two shards one acting as a primary and the other a replica.

What nodes are the shards on?

Create another database
Go back to the browser admin console and click on 'databases' in the nav menu. You should see the database you created. Redis Enterprise supports multi-tenancy: meaning you can create additional databases and will be allocated resources and managed independently.

Click on the '+' icon under your database name and create another database with replication (the trial license allows for four shards so you should have enough).

Now return to the terminal and run `rladmin>` status again. You should now see two databases with unique IDs, two endpoints and four shards total.

Where were the second database shards placed?

The cluster manager will place nodes according to available resources and ensure that the primary and replica shards are never on the same node.

This is a simple single primary shard with replica use-case but when we move on to scaling you will see how Redis Enterprise can maintain sharded databases across nodes with replication.

Stopping Environment

```
docker-compose down
```

Course: Running Redis at Scale

Unit 5: Scalability

5.1 Clustering in Redis

What is scaling?

In the next few chapters we'll talk about scaling. You'll learn what it means to scale, the different options for scaling, and when and how to scale your Redis deployment.

If you look up "Scalability" on Wikipedia, you'll see the following definition:

"Scalability is the property of a system to handle a growing amount of work by adding resources to the system."

The two most common scaling strategies are vertical scaling and horizontal scaling.

Vertical scaling, or also called "Scaling Up", means adding more resources like CPU or memory to your server.

Horizontal scaling, or "Scaling out", implies adding more servers to your pool of resources.

It's the difference between just getting a bigger server and deploying a whole fleet of servers.

Horizontal scaling in practice

Let's take an example. Suppose you have a server with 128 GB of RAM, but you know that your database will need to store 300 GB of data. In this case, you'll have two choices: you can either add more RAM to your server so it can fit the 300GB dataset, or you can add two more servers and split the 300GB of data between the three of them.

Hitting your server's RAM limit is one reason you might want to scale up, or out, but reaching the performance limit in terms of throughput, or operations per second, is also an indicator that scaling is necessary.

Since Redis is mostly single-threaded, Redis cannot make use of the multiple cores of your server's CPU for command processing.

But if we split the data between two Redis servers, our system can process requests in parallel, increasing the throughput by almost 200%. In fact, performance will scale close to linearly by adding more Redis servers to the system. This database architectural pattern of splitting data between multiple servers for the purpose of scaling is called **sharding**.

The resulting servers that hold chunks of the data are called **shards**.

How Redis sharding works

This performance increase sounds amazing, but we need a good technique for making it work: if we divide and distribute our data across two shards, which are just two Redis server instances, how will we know where to look for each key? We need a way to consistently map a key to a specific shard. There are many ways to do this, and different databases adopt different strategies. The one Redis uses is called "Algorithmic sharding". Here's how it works:

To find the shard for a given key, we compute a numeric hash value from the key name and modulo divide that numeric hash by the total number of shards.

Because we're using a deterministic hash function, the key “foo”, for example, will always end up on the same shard, as long as the number of shards stays the same.

Adding shards

But what happens if we want to increase our shard count even further, a process commonly called “resharding”? Let’s say we add one new shard so that our total number of shards is three.

When a client tries to read the key “foo” now, they will run the hash function and modulo divide by the number of shards, as before, but this time the number of shards is different and we’re modulo dividing with three instead of two. Understandably, the result may be different, pointing us to the wrong shard!

Resharding is a common issue with the algorithmic sharding strategy and can be solved by rehashing all the keys in the keyspace and moving them to the shard appropriate to the new shard count. This is not a trivial task, though, and it can require a lot of time and resources, during which the database will not be able to reach its full performance or might even become unavailable.

Solving the resharding problem

Redis uses a simple approach to solving the resharding problem by providing a logical abstraction that sits between a key and a shard: the **hash slot**.

Now, when hashing a key, we modulo by 16,384. The number that each key maps to is known as its hash slot.

Hash slots are assigned to shards by range. For example, imagine at 16-shard deployment, where slots 0-1000 are assigned to shard 0. In this case, any key that maps to a hash slot in the range 0-1000 will be stored on shard 0.

When we do need to reshard, we simply move hash slots from one shard to another, distributing the data as required across the different Redis instances.

Overall, **this hash slot strategy has the effect of dramatically reducing the number of keys that need to be relocated when we add a new shard**.

Redis Cluster

Now that we know what sharding is and how it works in Redis, we can finally introduce Redis Cluster. Redis Cluster provides a way to run a Redis installation where data is automatically split across multiple Redis servers, or shards. Redis Cluster also provides high availability. So, if you're deploying Redis Cluster, you don't need (or use) Redis Sentinel.

Redis Cluster can detect when a primary shard fails and promote a replica to a primary without any manual intervention from the outside. How does it do it? How does it know that a primary shard has failed, and how does it promote its replica to be the new primary shard?

We need to have replication enabled. Say we have one replica for every primary shard. If all our data is divided between three Redis servers, we would need a six-member cluster, with three primary shards and three replicas.

All 6 shards are connected to each other over TCP and constantly PING each other and exchange messages using a binary protocol. These messages contain information about which shards have responded with a PONG, so are considered alive, and which haven’t.

When enough shards report that a certain primary shard is not responding to them, they can agree to trigger a failover and promote the shard’s replica to become the new primary. How many

shards need to agree that a shard is offline before a failover is triggered?

Well, that’s configurable and you can set it up when you create a cluster, but there are some very important guidelines that you need to follow.

Avoiding the split brain scenario

If you have an even number of shards in the cluster, say six, and there’s a network partition that divides the cluster in two, you'll then have two groups of three shards. The group on the left side will not be able to talk to the shards from the group on the right side, so the cluster will think that they are offline and it will trigger a failover of any primary shards, resulting in a left side with all primary shards.

On the right side, the three shards will see the shards on the left as offline, and will trigger a failover on any primary shard that was on the left side, resulting in a right side of all primary shards.

Both sides, thinking they have all the primaries, will continue to receive client requests that modify data, and that is a problem, because maybe client A sets the key “foo” to “bar” on the left side, but a client B sets the same key’s value to “baz” on the right side.

When the network partition is removed and the shards try to rejoin, we will have a conflict, because we have two shards, holding different data, claiming to be the primary and we wouldn’t know which data is valid.

This is called a split brain situation, and is a very common issue in the world of distributed systems. A popular solution is to always keep an odd number of shards in your cluster, so that when you get a network split, the left and right group will do a count and see if they are in the bigger or the smaller group (also called majority or minority). If they are in the minority, they will not try to trigger a failover and will not accept any client write requests.

Here's the bottom line: **to prevent split-brain situations in Redis Cluster, always keep an odd number of primary shards and two replicas per primary shard.**

5.2 Exercise - Creating a Redis Cluster

Step 1

To create a cluster, we need to spin up a few empty Redis instances and configure them to run in cluster mode.

Here's a minimal configuration file for Redis Cluster:

```
#redis.conf file
port 7000
cluster-enabled yes
cluster-config-file nodes.conf
cluster-node-timeout 5000
appendonly yes
```

On the first line we specify the port on which the server should run, then we state that we want the server to run in cluster mode, with the `cluster-enabled` yes directive. `cluster-config-file` defines the name of the file where the configuration for this node is stored, in case of a server restart. Finally, `cluster-node-timeout` is the number of milliseconds a node must be unreachable for it to be considered in failure state.

Step 2

Let's create a cluster on your localhost with three primary shards and three replicas (remember, in production always use two replicas to protect against a split-brain situation). We'll need to bring up six Redis processes and create a `redis.conf` file for each of

them, specifying their port and the rest of the configuration directives above.

First, create six directories:

```
mkdir 7000 7001 7002 7003 7004 7005
```

Step 3

Then create the minimal configuration **`redis.conf`** file from above in each one of them, making sure you change the port directive to match the directory name. You should end up with the following directory structure:

```
- 7000
 - redis.conf
- 7001
 - redis.conf
- 7002
 - redis.conf
- 7003
 - redis.conf
- 7004
 - redis.conf
- 7005
 - redis.conf
```

Step 4

Open six terminal tabs and start the servers by going into each one of the directories and starting a Redis instance:

```
# Terminal tab 1
cd 7000
/path/to/redis-server ./redis.conf

# Terminal tab 2
cd 7001
/path/to/redis-server ./redis.conf
```

... and so on.

Step 5

Now that you have six empty Redis servers running, you can join them in a cluster:

```
redis-cli --cluster create 127.0.0.1:7000
127.0.0.1:7001 \
127.0.0.1:7002 127.0.0.1:7003 127.0.0.1:7004
127.0.0.1:7005 \
--cluster-replicas 1
```

Here we list the ports and IP addresses of all six servers and use the CREATE command to instruct Redis to join them in a cluster, creating one replica for each primary. Redis-cli will propose a configuration; accept it by typing yes. The cluster will be configured and joined, which means, instances will be bootstrapped into talking with each other.

Finally, you should see a message saying:

```
[OK] All 16384 slots covered
```

This means that there is at least a master instance serving each of the 16384 slots available.

Step 6

Let’s add a new shard to the cluster, which is something you might do when you need to scale.

First, as before, we need to start two new empty Redis instances (primary and its replica) in cluster mode. We create new directories 7006 and 7007 and in them we copy the same redis.conf file we used before, making sure we change the port directive in them to the appropriate port (7006 and 7007).

```
mkdir 7006 7007
cp 7000/redis.conf 7006/redis.conf
cp 7000/redis.conf 7007/redis.conf
```

```
# Update the port numbers in the files ./7006/
redis.conf and ./7007/redis.conf to 7006 and 7007,
respectively
```

Step 7

Let's start the Redis instances:

```
# Terminal tab 7
cd 7006
redis-server ./redis.conf

# Terminal tab 8
cd 7007
redis-server ./redis.conf
```

Step 8

In the next step we join the new primary shard to the cluster with the add-node command. The first parameter is the address of the new shard, and the second parameter is the address of any of the current shards in the cluster.

```
redis-cli --cluster add-node 127.0.0.1:7006
127.0.0.1:7000
```

Note: The Redis commands use the term "Nodes" for what we call "Shards" in this training, so a command named "add-node" would mean "add a shard".

Step 9

Finally we need to join the new replica shard, with the same add-node command, and a few extra arguments indicating the shard is joining as a replica and what will be its primary shard. If we don't specify a primary shard Redis will assign one itself.

We can find the IDs of our shards by running the cluster nodes command on any of the shards:

```
$ redis-cli -p 7000 cluster nodes
46a768cfeadb9d2aee91ddd882433a1798f53271
127.0.0.1:7006@17006 master - 0 1616754504000 0
connected
1f2bc068c7ccc9e408161bd51b695a9a47b890b2
127.0.0.1:7003@17003 slave
a138f48fe038b93ea2e186e7a5962fb1fa6e34fa 0
1616754504551 3 connected
5b4e4be56158cf6103ffa3035024a8d820337973
127.0.0.1:7001@17001 master - 0 1616754505584 2
connected 5461-10922
a138f48fe038b93ea2e186e7a5962fb1fa6e34fa
127.0.0.1:7002@17002 master - 0 1616754505000 3
connected 10923-16383
71e078dab649166dcbbcec51520742bc7a5c1992
127.0.0.1:7005@17005 slave
5b4e4be56158cf6103ffa3035024a8d820337973 0
1616754505584 2 connected
f224ecabedf39d1fffb34fb6c1683f8252f3b7dc
127.0.0.1:7000@17000 myself,master - 0
1616754502000 1 connected 0-5460
04d71d5eb200353713da475c5c4f0a4253295aa4
127.0.0.1:7004@17004 slave
f224ecabedf39d1fffb34fb6c1683f8252f3b7dc 0
1616754505896 1 connected
```

The port of the primary shard we added in the last step was 7006, and we can see it on the first line. It's id is `46a768cfeadb9d2aee91ddd882433a1798f53271.`

The resulting command is:

```
$ redis-cli -p 7000 --cluster add-node
127.0.0.1:7007 127.0.0.1:7000 --cluster-slave --
```

```
cluster-master-id
46a768cfeadb9d2aee91ddd882433a1798f53271
```

The flag cluster-slave indicates that the shard should join as a replica and **`--cluster-master-id 46a768cfeadb9d2aee91ddd882433a1798f53271`** specifies which primary shard it should replicate.

Step 10

Now our cluster has eight shards (four primary and four replica), but if we run the cluster slots command we'll see that the newly added shards don't host any hash slots, and thus - data. Let's assign some hash slots to them:

```
$ redis-cli -p 7000 --cluster reshard
127.0.0.1:7000
```

We use the command `reshard` and the address of any shard in the cluster as an argument here. In the next step we'll be able to choose the shards we'll be moving slots from and to.

The first question you'll get is about the number of slots you want to move. If we have 16384 slots in total, and four primary shards, let's get a quarter of all shards, so the data is distributed equally. 16384 ? 4 is 4096, so let's use that number.
The next question is about the receiving shard id; the ID of the primary shard we want to move the data to, which we learned how to get in the previous step, with the `cluster nodes` command.

Finally, we need to enter the IDs of the shards we want to copy data from. Alternatively, we can type "all" and the shard will move a number of hash slots from all available primary shards.

```
$ redis-cli -p 7000 --cluster reshard
127.0.0.1:7000
....
....

How many slots do you want to move (from 1 to
16384)? 4096
What is the receiving node ID?
46a768cfeadb9d2aee91ddd882433a1798f53271
Please enter all the source node IDs.
 Type 'all' to use all the nodes as source nodes
for the hash slots.
 Type 'done' once you entered all the source nodes
IDs.
Source node #1: all

Ready to move 4096 slots.
 Source nodes:
           M:
f224ecabedf39d1fffb34fb6c1683f8252f3b7dc
127.0.0.1:7000
          slots:[0-5460] (5461 slots) master
          1 additional replica(s)
           M:
5b4e4be56158cf6103ffa3035024a8d820337973
127.0.0.1:7001
          slots:[5461-10922] (5462 slots) master
          1 additional replica(s)
           M:
a138f48fe038b93ea2e186e7a5962fb1fa6e34fa
127.0.0.1:7002
          slots:[10923-16383] (5461 slots) master
          1 additional replica(s)
 Destination node:
           M:
46a768cfeadb9d2aee91ddd882433a1798f53271
127.0.0.1:7006
          slots: (0 slots) master
          1 additional replica(s)
 Resharding plan:
```

```
           Moving slot 5461 from
5b4e4be56158cf6103ffa3035024a8d820337973
           Moving slot 5462 from
5b4e4be56158cf6103ffa3035024a8d820337973

Do you want to proceed with the proposed reshard
plan (yes/no)?
Moving slot 5461 from 127.0.0.1:7001 to
127.0.0.1:7006:
Moving slot 5462 from 127.0.0.1:7001 to
127.0.0.1:7006:
Moving slot 5463 from 127.0.0.1:7001 to
127.0.0.1:7006:
....
....
....
```

Once the command finishes we can run the cluster slots command again and we'll see that our new primary and replica shards have been assigned some hash slots:

```
$ redis-cli -p 7000 cluster slots
```

5.3 Exercise - Using redis-cli with a Redis Cluster

When you use `redis-cli` to connect to a shard of a Redis Cluster, you are connected to that shard only, and cannot access data from other shards. If you try to access keys from the wrong shard, you will get a `MOVED` error.

There is a trick you can use with redis-cli so you don't have to open connections to all the shards, but instead you let it do the connect and reconnect work for you. It's the redis-cli cluster support mode, triggered by the -c switch:

```
$ redis-cli -p 7000 -c
```

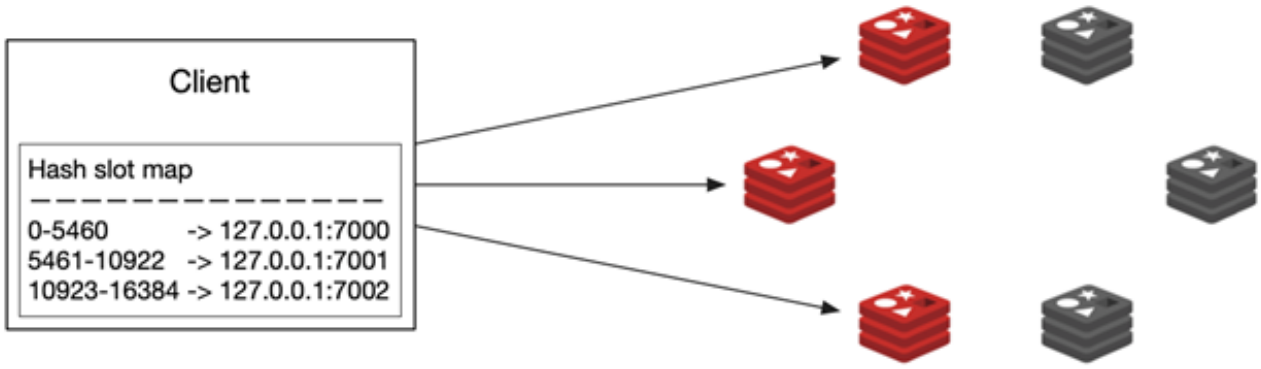
When in cluster mode, if the client gets an `(error) MOVED 15495 127.0.0.1:7002` error response from the shard it's connected to, it will simply reconnect to the address returned in the error response, in this case `127.0.0.1:7002`.

Now it's your turn: use redis-cli cluster mode to connect to your cluster and try accessing keys in different shards. Observe the response messages.

5.4 Redis Cluster and Client Libraries

To use a client library with Redis Cluster, the client libraries need to be cluster-aware. Clients that support Redis Cluster typically feature a special connection module for managing connections to the cluster. The process that some of the better client libraries follow usually goes like this:

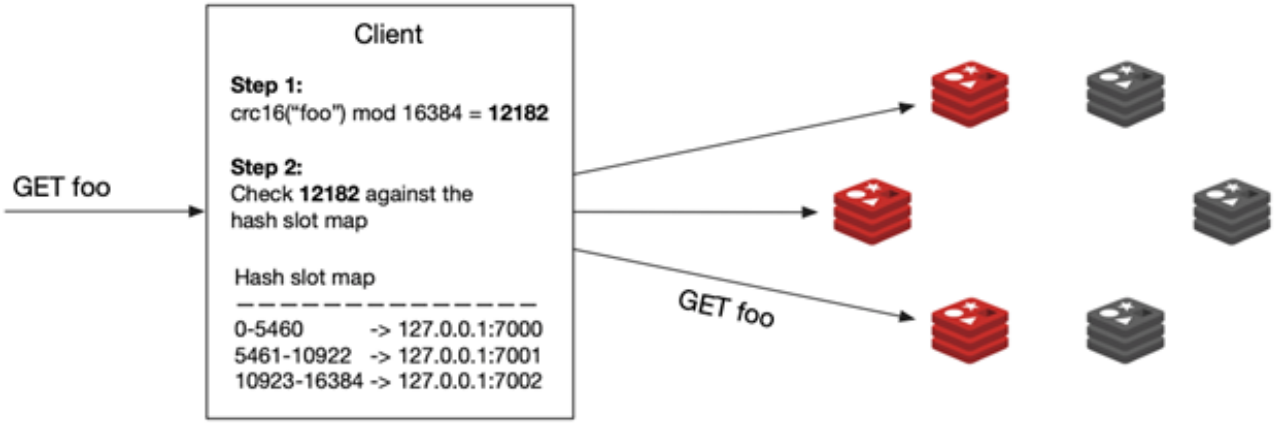
The client connects to any shard in the cluster and gets the addresses of the rest of the shards. The client also fetches a mapping of hash slots to shards so it can know where to look for a key in a specific hash slot. This hash slot map is cached locally.



When the client needs to read/write a key, it first runs the hashing function (crc16) on the key name and then modulo divides by 16384, which results in the key's hash slot number.

In the example below the hash slot number for the key "foo" is 12182. Then the client checks the hashlot number against the hash slot map to determine which shard it should connect to. In our example, the hash slot number 12182 lives on shard `127.0.0.1:7002`.

Finally, the client connects to the shard and finds the key it needs to work with.



If the topology of the cluster changes for any reason and the key has been moved, the shard will respond with an `(error) MOVED 15495 127.0.0.1:7006` error, returning the address of the new shard responsible for that key. This indicates to the client that it needs to re-query the cluster for its topology and hash slot allocation, so it will do that and update its local hash slot map for future queries.

Not every client library has this extra logic built in, so when choosing a client library, make sure to look for ones with cluster support.

Another detail to check is if the client stores the hash slot map locally. If it doesn't, and it relies on the `(error) MOVED` response to get the address of the right shard, you can expect to have a

much higher latency than usual because your client may have to make two network requests instead of one for a big part of the requests.

Examples of clients that support Redis cluster:

Java: Jedis, Lettuce
.NET: StackExchange.Redis
Go: Radix, go-redis/redis
Node.js: ioredis
Python: redis-py
https://redis.io/clients

5.5 Scaling in Redis Enterprise

Up to this point, you've seen how open source Redis scales with Redis Cluster. As your Redis deployment matures, you may encounter administrative challenges with Redis Cluster. For example:

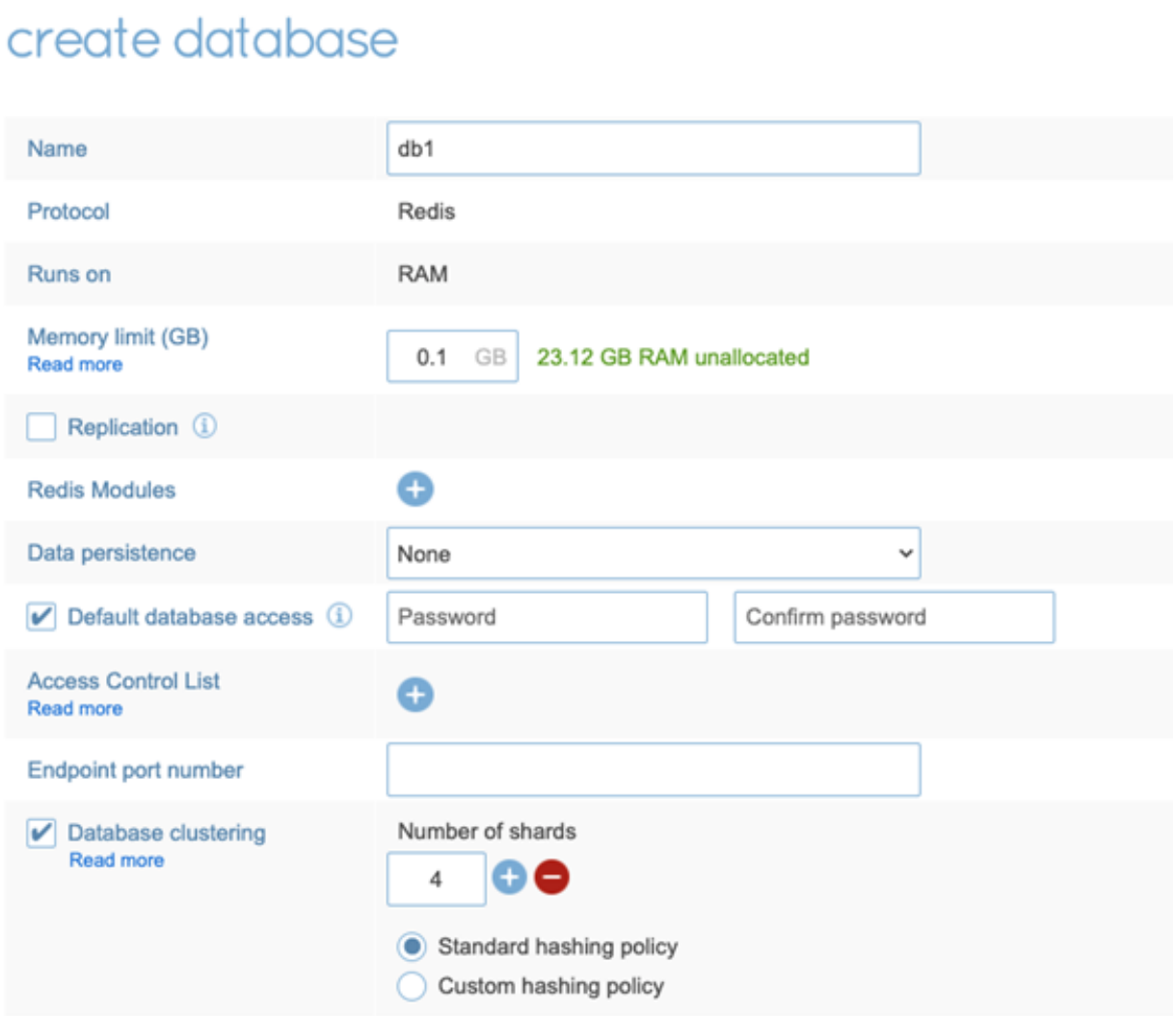
- How can we better utilize our compute resources?
- What happens when we outgrow our current Redis deployment?
- How do we provision and manage multiple Redis databases for our many internal customers?

In reality, you'll need a dedicated ops team to handle the many moving parts in a typical Redis Cluster deployment. Redis Enterprise was designed to effectively use resources and simplify the administration of multiple Redis databases (among other things!). Let's take a closer look here.

Resource Utilization
As mentioned in another unit, Redis Enterprise was designed first and foremost for a DBaaS (database-as-a-service) which means that resource utilization is critical. Anyone who has made the transition to the cloud has probably seen the cost of underutilized resources.

Clustering
We've already covered how clustering works with Redis in general, but let's take a look at how Redis Enterprise simplifies this.

A database can be clustered easily through the admin console by simply enabling clustering and deciding how many shards you need. In this example, a database named 'db1' will be created with four shards:



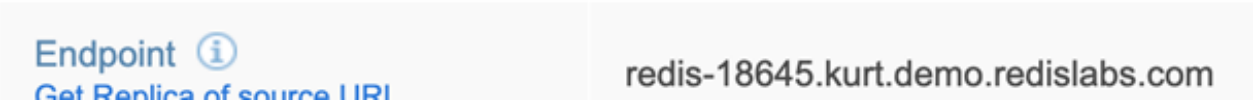
Behind the scenes, Redis Enterprise will create a Redis database with four shards, each containing unique hash slots.

SHARDS:

DB:ID	NAME	ID	NODE	ROLE	SLOTS	USED_MEMORY	STATUS
db:1	db1	redis:1	node:1	master	0-4095	1.94MB	OK
db:1	db1	redis:2	node:1	master	4096-8191	1.94MB	OK
db:1	db1	redis:3	node:1	master	8192-12287	1.94MB	OK
db:1	db1	redis:4	node:1	master	12288-16383	1.94MB	OK

Redis Enterprise also utilizes a proxy, which handles all the hash based sharding and communication to these shards while providing a single endpoint for clients to communicate with.

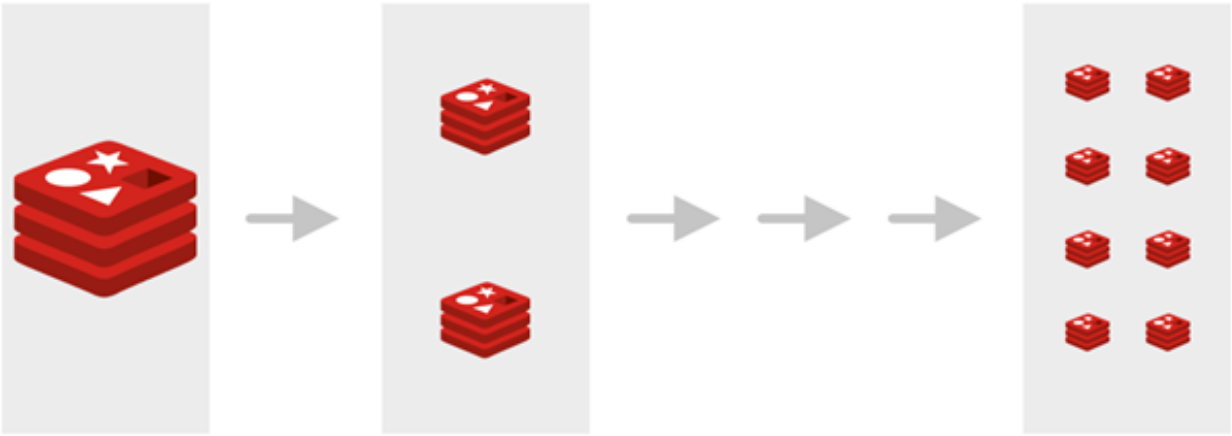
This means that you do not have to implement the complex Redis Cluster connection logic!



For example, his clustered database is exposed outside the Redis Enterprise cluster as a single endpoint, making it much easier for your clients to connect.

Scaling Up

This architecture allows Redis Enterprise to scale up. Shards, each with their own hash slots, will be deployed efficiently as the underlying cluster resources permit. The proxy will take care of connecting to the correct shard on behalf of the client.



Notice the screen capture of the shard details again. Look at the NODE column and you'll see that that all of these Redis shards are running on the same machine: node 1.

SHARDS:

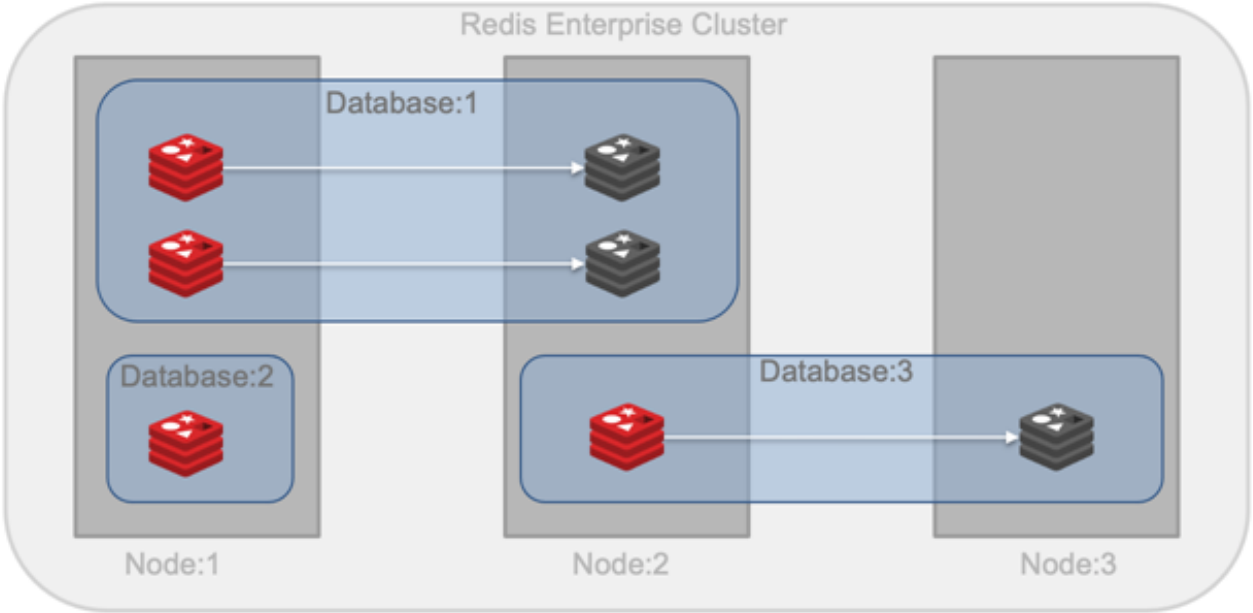
DB:ID	NAME	ID	NODE	ROLE	SLOTS	USED_MEMORY	STATUS
db:1	db1	redis:1	node:1	master	0-4095	1.94MB	OK
db:1	db1	redis:2	node:1	master	4096-8191	1.94MB	OK
db:1	db1	redis:3	node:1	master	8192-12287	1.94MB	OK
db:1	db1	redis:4	node:1	master	12288-16383	1.94MB	OK

Given the (mostly) single-threaded nature of Redis, this allows you to utilize more CPU on each node!

Multi-Tenancy

As mentioned, Redis Enterprise lets you create multiple databases across the same cluster of resources This makes it easy to support multiple teams or use cases that require separate Redis databases with different configurations.

For example, this cluster diagram depicts three different databases deployed across three nodes (machines), each with their own requirements around scaling and high availability.

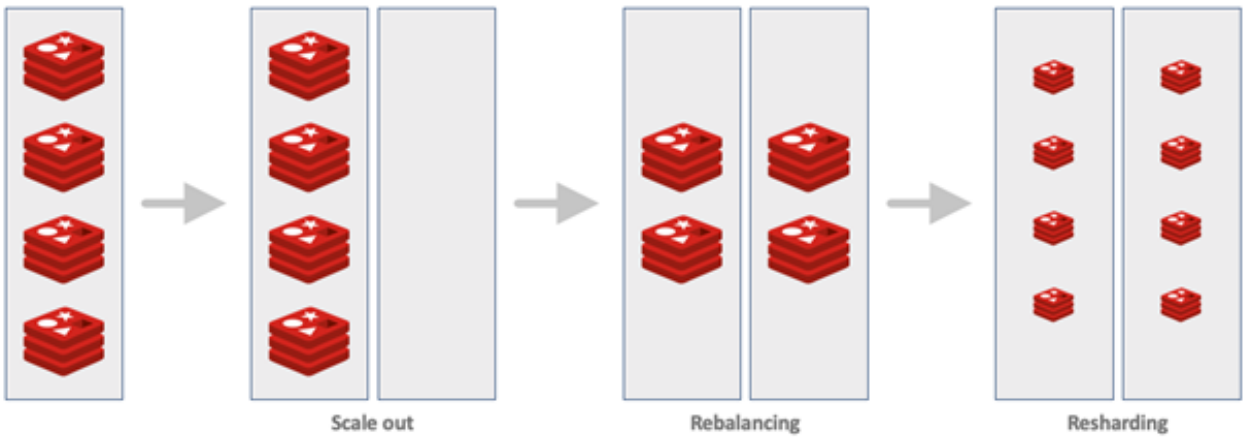


Increasing Capacity and Performance

So what happens when you outgrow your simple Redis deployment? What if there is a demand for larger Redis databases or more performance?

Scaling Out

As we have seen, Redis Cluster lets you partition your database across multiple Redis processes. You also saw how Redis Enterprise handles this database clustering in its management layer. And we've shown how you can also scale out your database on Redis Enterprise across multiple nodes while still maintaining a single endpoint.



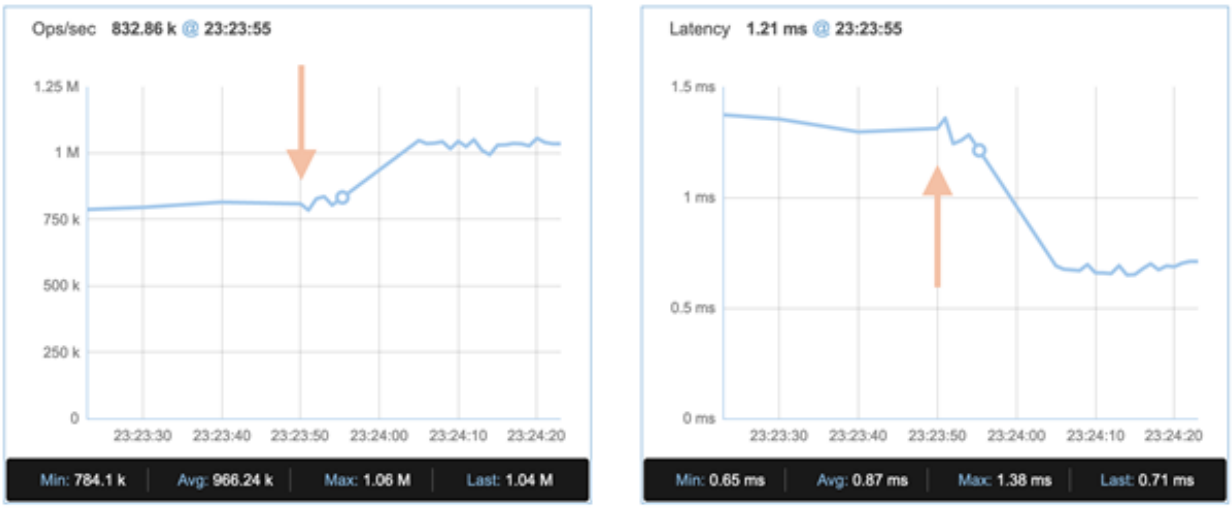
This means that as long as there are enough resources in your Redis Enterprise cluster, you can continue to increase the scale of a database. Here is an example of another database with 8 shards spread across three nodes. You can see that each node is handling multiple shards:

```
SHARDS:
DB:ID      NAME      ID          NODE        ROLE        SLOTS           USED_MEMORY   STATUS
db:2       db2       redis:5     node:1      master      0-2047          3.87MB        OK
db:2       db2       redis:6     node:2      master      4096-6143       3.83MB        OK
db:2       db2       redis:7     node:3      master      8192-10239      4.02MB        OK
db:2       db2       redis:8     node:1      master      12288-14335     3.87MB        OK
db:2       db2       redis:9     node:2      master      14336-16383     3.83MB        OK
db:2       db2       redis:10    node:3      master      2048-4095       3.83MB        OK
db:2       db2       redis:11    node:1      master      10240-12287     3.83MB        OK
db:2       db2       redis:12    node:2      master      6144-8191       3.9MB         OK
```

Impacting Throughput

Scaling out through Redis Enterprise clearly allows you to increase capacity, but how does scaling out affect performance? Just by getting more Redis shards involved, you will be adding to the overall CPU resources for a database.

Here is a simple example of a database under load and what happens to its performance after it scales. You can see the throughput goes up and latency goes down (the orange arrow indicates when sharding was enabled in Redis Enterprise):

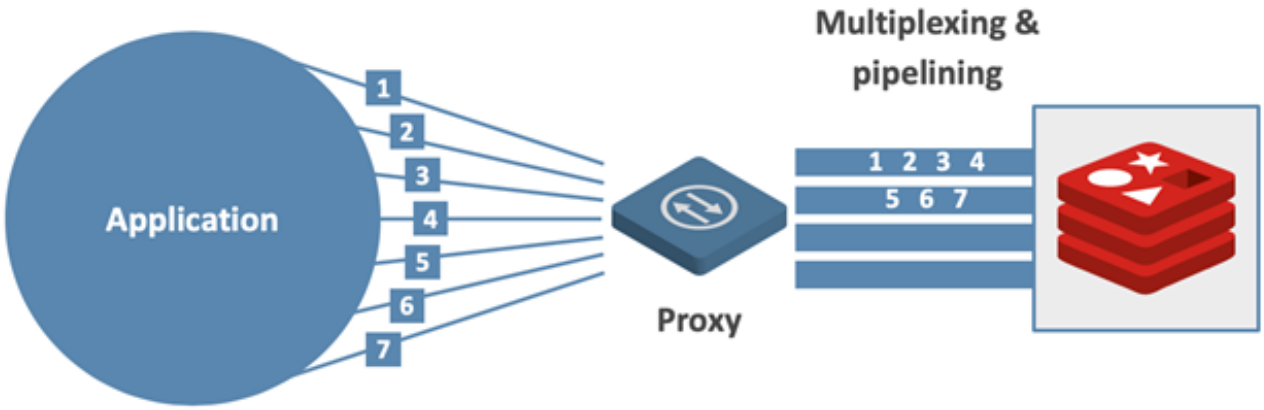


Seamless Scaling

In the diagram in the last section you can see the throughput and latency of a database under load as it is scaling. Notice that the traffic never completely stops during the scaling event. Redis Enterprise supports scaling without downtime. An existing clustered database can be increased while clients are connected. The Redis Enterprise cluster manager scales out and configures the endpoint with the new cluster topology behind the scenes so clients do not need to go through a re-discovery as the endpoint has not changed.

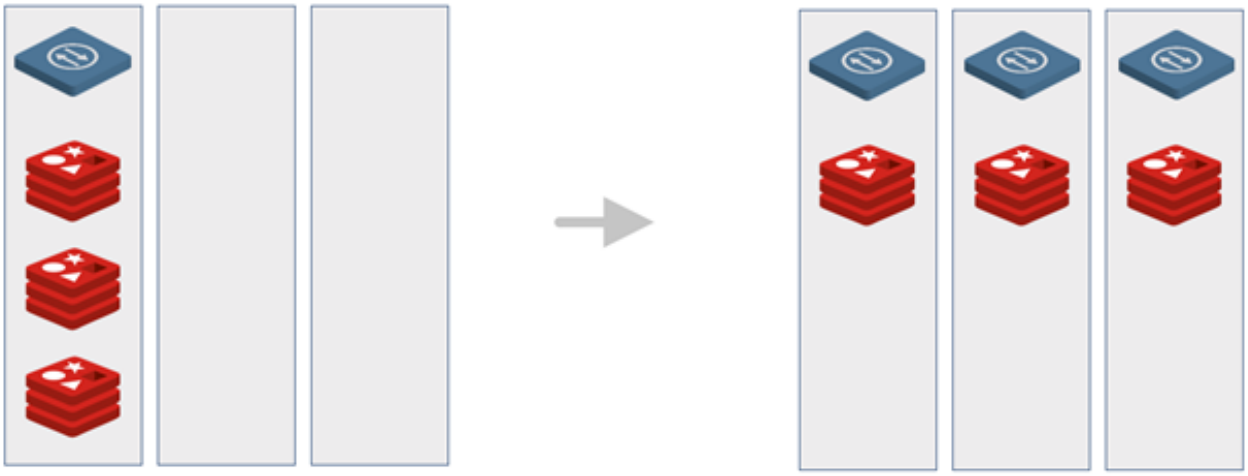
Scaling the Proxy

In addition to scaling out the database, the Redis Enterprise proxy provides significant performance benefits. As mentioned before, this proxy offloads all connection handling from the Redis processes by multiplexing while maintaining a few persistent connections to the shards behind the scenes. The proxy also uses pipelining when sending requests to the shards to improve performance.



The proxy is multi-threaded and can be scaled up by the Redis Enterprise cluster manager. Additional tuning can be done if more cores are available and are not being used by the defaults.

The proxy can also scale out across multiple nodes for each database.



5.6 Exercise - Seamless Scaling in Redis Enterprise

Clone this repo if you have not already: https://github.com/redislabs-training/exercises-scaling-redis

Change into the 'scaling-seamlessly' directory.

Requirements

- docker
- docker-compose
- internet connection

Starting Environment

```
docker-compose up -d
```

Getting started

After the `docker-compose -d` command has successfully started up the Redis Enterprise docker container we need to set up a Redis Enterprise cluster. Since you have already gone through the individual steps to do this in the High Availability exercise there is a setup script provided.
Run from this directory:

```
./setup.sh
```

If you wish to go through the setup manually you can go back to the High Availability exercise and go through the steps there or open the setup.sh and view the commands that are being run.

Once that completes with a successful message you should be able to open a browser:

https://localhost:18443/

Redis Enterprise creates self-signed certs in the beginning so a secure connection is always created. Since they are not CA signed certificates you will need to allow your browser to accept them.

Depending on your browser and version you may have an 'advanced' option that allows you to accept and continue/proceed. Some Chrome browser versions will not have this option, but you can bypass this warning by just typing 'thisisunsafe' and it will automatically continue through to the site.

Once you bypass the browser warning you should see a login screen. Use these credentials

username: admin@redis.com
password: redis123

Create the database
After a successful login you should be presented with the prompt to create a new database. Select 'redis database,' 'Runs on ram' and 'Single Region' and click 'Next.'

You should now have a screen to enter the database details. Let's keep it simple to start with by entering a name and then click the 'Activate' button. Redis Enterprise will create a single sharded database.

Open a terminal to and connect to the rladmin utility and view the cluster status to see the details.

```
docker-compose exec re1 bash
```

During the Redis Enterprise exercise on High Availability you used rladmin in interactive mode, but you can also pass commands and options in directly as args.

```
:/opt$ rladmin status shards
```

This will show the `SHARDS` section of the status. You should just have a single shard for the database. Now get the databases section so you can find the endpoint, particularly the port.

```
:/opt$ rladmin status databases
```

Look in the **`ENDPOINT`** column and copy the port that was generated for this database. Since we don't have the full features of Redis Enterprise with DNS, etc. in this dockerized environment we

will just call the database using it's external port. The request will still go through the proxy.

Create Traffic
Now let's use that port in **`memtier_benchmark`** a Redis benchmarking utility. Just replace the port from the example with your own. You should see the benchmark client begin to send traffic to your database:

```
memtier_benchmark -p 19100 -x 1000
[RUN #1] Preparing benchmark client...
[RUN #1] Launching threads now...
```

Scale The Database
Go back to your browser and reload the database page. You may need to login again if the session has expired. View the 'metrics' tab and verify traffic is being sent to your database.

Go back to the configuration view and click 'Edit' at the bottom to edit the database. Enable the checkbox for 'Database clustering' and enter four shards and click 'Update' at the bottom. The Redis Enterprise cluster manager will update the database using its state machine to add the additional Redis processes, re-shard, re-balance and re-bind the endpoint.

Open the 'metrics' view again, your database should still be taking client requests.

NOTE: With the limited resources of this dockerized environment there may be performance bumps. This exercise is not intended to show an increase in performance just the ease of managing the scaling process.

Stop traffic
You can use `ctrl-c` in the terminal to stop the traffic.

Stopping Environment
```
docker-compose down
```

Unit 6

Course: Running Redis at Scale

Unit 6: Observability

6.1 Observability in Redis

The last thing you want to do after successfully deploying and scaling Redis for use by your applications is to be stuck on the weekend because performance is down or the service is completely unavailable!

So to start, let's look at some of those difficult questions so you don't find yourself in that situation.

Is my Redis deployment up and running?

A simple utility command like `PING` is an extremely simple yet effective means of answering this question.

```
$ redis-cli -p ping
PONG
```

Though the `PING` command is less costly to run than `INFO` there are uptime stats returned by it that can add further data to this answer.

```
$ redis-cli info | grep uptime
uptime_in_seconds:74718
uptime_in_days:0
```

Gathering stats from the `INFO` command will be explored further on in this unit.

Is my Redis deployment at capacity?

A comparison of the current memory usage from the INFO stats and the configured max memory would give a clear idea how much capacity has been used and how much is remaining.

```
$ redis-cli INFO |grep used_memory:
$ redis-cli config get maxmemory
```

You can look at evictions and expirations to combine with usage to see if memory is being freed enough to keep up.

```
$ redis-cli INFO stats |grep evicted_keys
$ redis-cli INFO stats | grep expired_keys
```

A cache hit/miss ratio would help in determining the current capacity levels as well:

```
$ redis-cli INFO stats |grep keyspace
keyspace_hits:6281381
keyspace_misses:62754834
```

You will get a chance to see later in this unit how Redis Enterprise dashboards provide these and many other metrics already calculated.

Is my Redis deployment accessible?

One quick way to determine this is to look at your current client connections and make sure they are above an expected minimum threshold.

```
$ redis-cli info clients | grep clients
connected_clients:1
blocked_clients:0
tracking_clients:0
clients_in_timeout_table:0
```

An overview of stats from the `INFO` command will be explored further on in this unit.

Is my Redis deployment performing as expected?

The in-memory performance is one of the main drivers to deploy and use Redis so this metric must be monitored with a clear idea of what is the expected performance.

One simple way to get the data to answer this question is using the Redis CLI client to pull real-time latency data from your Redis database.

```
$ redis-cli --latency
min: 1, max: 17, avg: 4.03 (927 samples)
```

We can also use the Redis Slowlog to determine any commands to look for performance issues.

We will explore the Latency Framework and Slowlog built into Redis in sections later on in this unit. You will also get a chance to see how they are available in Redis Enterprise via it's built-in dashboards.

What happened to my Redis deployment?

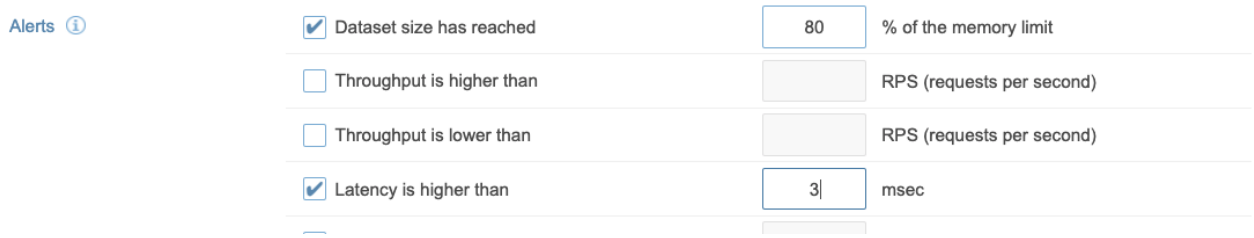
This is one of those questions you hope you don't have to answer because it means something is currently broken and we need to fix it or we are doing a post-mortem on an event. It still needs to be done.

While there are several built-in utilities that will aid in your troubleshooting, you may end up having to go back to the logs. We will explore how to set up logging for Redis and take a look at how and where the Redis Enterprise processes are logged.

How can I find out all of this ahead of time?

Wouldn't it be great if you did not have to answer the question above as much?

This unit will provide some examples for building alerts based on Redis stats. Also you will see how Redis Enterprise comes with alerts that enable you to get warned when capacity has hit a certain percentage of the memory limit or when latency is higher than a certain amount of MS.



Now let's look at where you can find the data hinted at above and what some of those utilities look like in action.

6.2 Data Points in Redis

Redis has a few ways of getting data that can be viewed through redis-cli.

Redis INFO command

Running the `INFO` command provides many of the metrics available in a single view.

```
redis> info
# Server
redis_version:6.0.1
redis_git_sha1:00000000
redis_git_dirty:0
redis_build_id:e02d1d807e41d65
redis_mode:standalone
os:Linux 4.19.121-linuxkit x86_64
...
```

There are several sections that can be pulled individually. For example, if you wanted to just get the ‘clients’ section you can pass that section as an argument to the info command.

```
redis> info clients
# Clients
connected_clients:1
client_recent_max_input_buffer:2
client_recent_max_output_buffer:0
blocked_clients:0
tracking_clients:0
clients_in_timeout_table:0
```

Sections:

Server: the current Redis server info

Metrics of note:

- `redis_version`
- `process_id`
- `config_file`
- `uptime_in_seconds`
- `uptime_in_days`

Clients: available data on clients connected or failed connections
Metrics of note:

- `connected_clients`
- `blocked_clients`

Memory: memory usage and stats
Metrics of note:

- `used_memory`
- `mem_fragmentation_ratio`

Persistence: RDB or AOF metrics
Metrics of note:

- `rdb_last_save_time`
- `rdb_changes_since_last_save`
- `aof_rewrite_in_progress`

Stats: some general statistics
Metrics of note:

- `keyspace_hits`
- `keyspace_misses`
- `expired_keys`
- `evicted_keys`
- `instantaneous_ops_per_sec`

Replication: replication data including primary/replica identifiers and offsets
Metrics of note:

- `master_link_down_since`
- `connected_slaves`
- `master_last_io_seconds_ago`

CPU: compute consumption stats

Metrics of note:

- `used_cpu_sys`
- `used_cpu_user`

Modules: data from any loaded modules
Metrics of note (per module):

- `ver`
- `options`

Cluster: whether cluster is enabled
Metric of note:

- `cluster_enabled`

Keyspace: keys and expiration data
Metrics of note (per db):

- `keys`
- `expires`
- `avg_ttl`

The output can be read from the results or piped into a file.

```
> redis-cli info stats > redis-info-stats
```

This could be done at intervals and consumed by a local or third party monitoring service.

Note that some of the data in `INFO` are going to be static data like the Redis `version` that won't change until an update is made and data that will be changing with use like `keyspace_hits ÷ keyspace_misses`. The latter could be taken to compute a hit ratio and observed as a long term analytic. The replication section filed `master_link_down_since` could be a metric to connect an alert.

Some examples of possible alerts that could be setup for a given metric:

Metric Example Alert

`uptime_in_seconds` < 300 seconds: to ensure the server is staying up

`connected_clients` < minimum number of expected application connections

`master_link_down_since` > 30 seconds: replication should be operational

`rdb_last_save_time` > maximum acceptable interval without taking a snapshot

NOTE: This is not an exhaustive list, but just to give you an idea of how the metrics in `INFO` could be used.

Latency and stats data via redis-cli options

The redis-cli client has some built-in options that allow you to pull some real-time latency and stats data.

Note: these are not available as commands from Redis but as options in redis-cli.

Latency options:

Continuously sample latency

```
$ redis-cli --latency
min: 1, max: 17, avg: 4.03 (927 samples)
```

The raw or csv output flag can be added

```
$ redis-cli --latency --csv
1,4,1.94,78
```

In order to sample for longer than one second you can use latency-history which has a default interval of 15 seconds but can specified using the `-i` param.

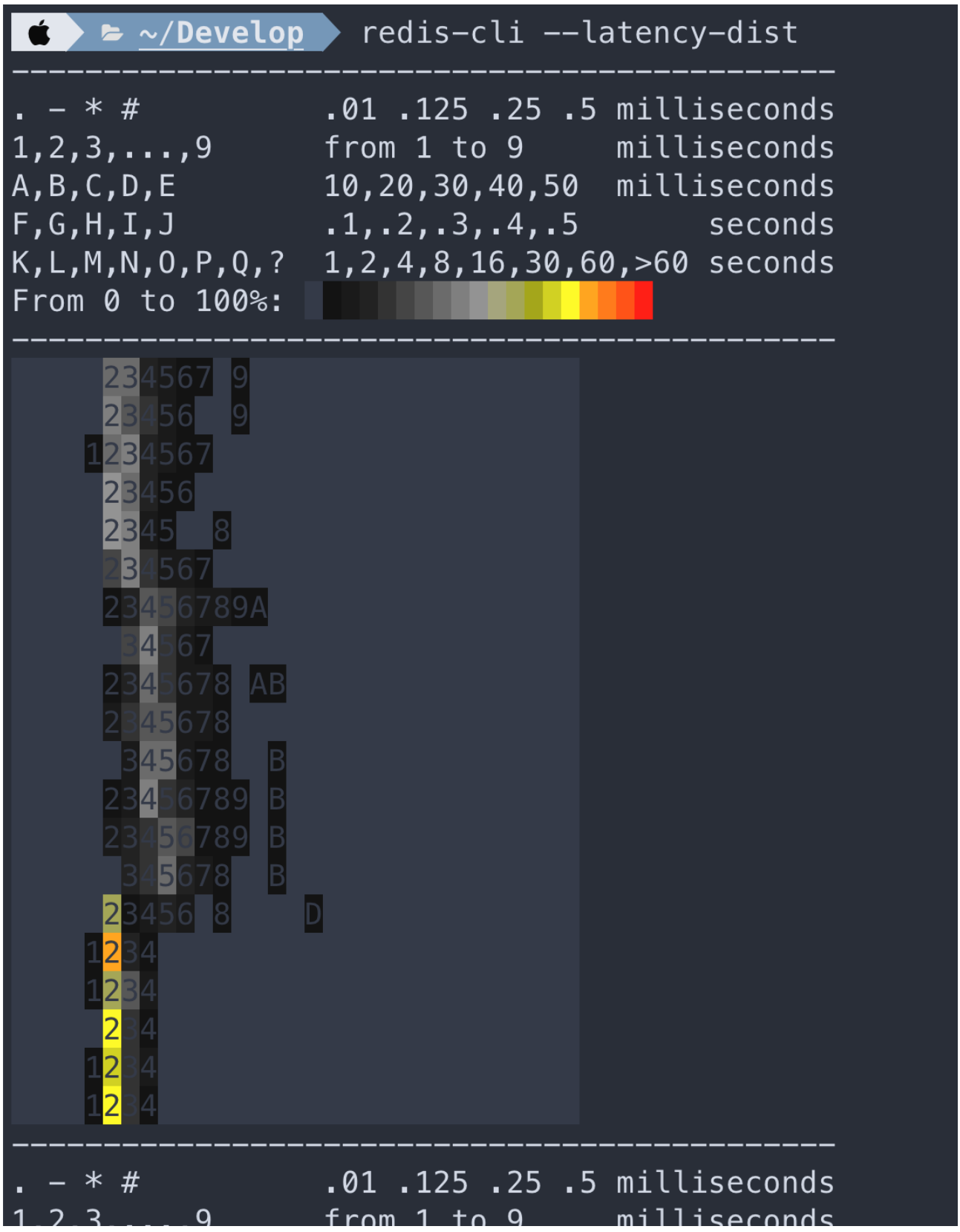
```
$ redis-cli --latency-history -i 60
min: 1, max: 30, avg: 4.84 (328 samples)
```

This can also be combined with the csv or raw output format flag.

```
$ redis-cli --latency-history -i 60 --csv
13,13,13.00,1
5,13,9.00,2
3,13,7.00,3
3,13,6.00,4
3,13,5.60,5
2,13,5.00,6
2,13,5.43,7
2,13,5.62,8
2,13,5.22,9
2,13,5.00,10
1,13,4.64,11
…
```

Both of these could be piped to a file as well.

The latency-dist option shows latency as a spectrum. The default interval is one second but can be changed using the `-i` param.



Stats option:

Get rolling stats from the server using the stat flag.

```
$ redis-cli --stat
------- data ------ --------------------- load
-------------------- - child -
keys       mem     clients blocked
requests            connections
4          9.98M   51      0       8168035
(+0)        4132
4          9.98M   51      0       8181542
(+13507)    4132
4          9.98M   51      0       8196100
(+14558)    4132
4          9.98M   51      0       8209794
(+13694)    4132
4          9.98M   51      0       8223420
(+13626)    4132
4          9.98M   51      0       8236624
(+13204)    4132
4          9.98M   51      0       8251376
(+14752)    4132
4          9.98M   51      0       8263417
(+12041)    4182
4          9.98M   51      0       8276781
(+13364)    4182
4          9.90M   51      0       8289693
(+12912)    4182
```

Memory stats

Redis includes a `MEMORY` command that includes a subcommand to get stats.

```
redis> memory stats
 1) "peak.allocated"
 2) (integer) 11912984
 3) "total.allocated"
 4) (integer) 8379168
 5) "startup.allocated"
 6) (integer) 5292168
 7) "replication.backlog"
```

```
 8) (integer) 0
 9) "clients.slaves"
10) (integer) 0
11) "clients.normal"
12) (integer) 16986
13) "aof.buffer"
14) (integer) 0
```

These values are available in the `INFO MEMORY` command as well, but here they are returned in a typical Redis RESP Array reply.

There is also a `LATENCY DOCTOR` subcommand with an analysis report of the current memory metrics.

Latency Monitoring

As we know Redis is fast and as a result is often used in very extreme scenarios where low latency is a must. Redis has a feature called Latency Monitoring which allows you to dig into possible latency issues. Latency monitoring is composed of the following conceptual parts:

- Latency hooks that sample different latency sensitive code paths.
- Time series recording of latency spikes split by different events.
- A reporting engine to fetch raw data from the time series.
- Analysis engine to provide human readable reports and hints according to the measurements.

By default this feature is disabled because most of that time it is not needed. In order to enable it you can update the threshold time in milliseconds that you want to monitor in your Redis configuration. Events that take longer than the threshold will be logged as latency spikes. If the requirement is to identify all events blocking the server for a time of 10 milliseconds or more I should set this threshold configuration accordingly.

```
latency-monitor-threshold 10
```

If the debugging session is intended to be temporary the threshold can be set via redis-cli.

```
redis> CONFIG SET latency-monitor-threshold 10
```

To disable the latency framework the threshold should be set back to 0.

```
redis> CONFIG SET latency-monitor-threshold 0
```

The latency data can be viewed using the `LATENCY` command with it's subcommands:

`LATENCY LATEST` - latest samples for all events

`LATENCY HISTORY` - latest time series for a given event

`LATENCY RESET` - resets the time series data

`LATENCY GRAPH` - renders an ASCII-art graph

`LATENCY DOCTOR` - analysis report

In order to make use of these commands you need to make yourself familiar with the different events that the latency monitoring framework is tracking. (taken from https://redis.io/topics/latency-monitor)

Event Description
`command` regular commands
`fast-command` O(1) and O(log N) commands
`fork` the fork(2) system call
`rdb-unlink-temp-file` the unlink(2) system call

`aof-write` writing to the AOF - a catchall event fsync(2) system calls
`aof-fsync-always` the fsync(2) system call when invoked by the appendfsync allways policy
`aof-write-pending-fsync` the fsync(2) system call when there are pending writes
`aof-write-active-child` the fsync(2) system call when performed by a child process
`aof-write-alone` the fsync(2) system call when performed by the main process
`aof-fstat` the fstat(2) system call
`aof-rename` the rename(2) system call for renaming the temporary file after completing BGREWRITEAOF
`aof-rewrite-diff-write` writing the differences accumulated while performing BGREWRITEAOF
`active-defrag-cycle` the active defragmentation cycle
`expire-cycle` the expiration cycle
`eviction-cycle` the eviction cycle
`eviction-del` deletes during the eviction cycle

For example, you can use the `LATENCY LATEST` subcommand and you may see some data like this:

```
redis> latency latest
1) 1) "command"
   2) (integer) 1616372606
   3) (integer) 600
   4) (integer) 600
2) 1) "fast-command"
   2) (integer) 1616372434
   3) (integer) 12
   4) (integer) 12
```

The results of this command provide the timestamp, latency latency and max latency for this event. Utilizing the events table above I can see we had latency spikes for a regular command with the latest and max latency of 600 MS while a O(1) or O(log N) command had a latency spike of 12 MS.

Some of the latency commands require a specific event be passed.

```
redis> latency graph command
command - high 600 ms, low 100 ms (all time high
600 ms)
--------------------------------------------------
----------------------------
    _##
  o|||
 o||||
_#|||||

3222184
05308ss
sssss
```

While the cost of enabling latency monitoring is near zero and memory requirements are very small it will raise your baseline memory usage so if you are getting the required performance out of Redis there is no need to leave this enabled.

Monitoring Tools

There are many open source monitoring tools and services to visualize your Redis metrics some of which also provide alerting capabilities.

One example of this is the Redis Data Source for Grafana. It is a Grafana plug-in that allows users to connect to the Redis database and build dashboards to easily observe Redis data. It provides an out-of-the-box predefined dashboard but also lets you build customized dashboards tuned to your specific needs.

6.3 Exercise - Getting Redis Stats

Clone this repo if you have not already: https://github.com/redislabs-training/exercises-scaling-redis

Change into the ‘observability-’ directory.

Requirements

- docker
- docker-compose
- internet connection

Starting Environment

```
docker-compose up -d
```

Connect to the Environment

In a terminal run this command to get a shell prompt inside the running docker container:

```
docker-compose exec redis_stats bash
```

Generate load

A simple way to to generate some load is to open another terminal and run:

```
docker-compose exec redis_stats redis-benchmark
```

Info

Since most of the stats data comes from the INFO command you should first run this to view that there.

```
# redis-cli INFO
```

Try piping this output to a file.

Memory usage

Since we generally recommend setting the `maxmemory` size, it is possible to calculate the percentage of memory in use and alert based on results of the maxmemory configuration value and the `used_memory` stat.

First set the `maxmemory`.

```
# redis-cli config set maxmemory 100000
```

Then you can pull the two data points to see how that could be used to calculate memory usage.

```
# redis-cli INFO |grep used_memory:
# redis-cli config get maxmemory
```

Client data

You can pull the clients section of the `INFO` command:

```
# redis-cli info clients
```

or maybe a particular metric you would want to track:

```
# redis-cli info client | grep connected_clients
```

Stats section

Use redis-cli to list the full 'stats' section.

Hit ratio

A cache hit/miss ratio could be generated using two data points in the stats section.

```
# redis-cli INFO stats |grep keyspace
```

Evicted keys

Eviction occurs when redis has reached its maximum memory and maxmemory-policy in `redis.conf` is set to something other than `volatile-lru`.

```
# redis-cli INFO stats | grep evicted_keys
```

Expired keys

It is a good idea to keep an eye on the expirations to make sure redis is performing as expected.

```
# redis-cli INFO stats | grep expired_keys
```

Keyspace

The following data could be used for graphing the size of the keyspace as a quick drop or spike in the number of keys is a good indicator of issues.

```
# redis-cli INFO keyspace
```

Workload (connections received, commands processed)

The following stats are a good indicator of workload on the Redis server.

```
# redis-cli INFO stats |egrep "^total_"
```

6.4 Identifying Issues

Besides the metrics from the data points from info, memory and the latency framework in the sections above, you may need to pull data from other sources when troubleshooting.

Availability

The redis server will respond to the `PING` command when running properly

```
redis-cli -h redis.example.com -p 6379 PING
PONG
```

Slowlog

Redis Slow Log is a system to log queries that exceed a specific execution time which does not include I/0 operations like client communication. It is enabled by default with two configuration parameters.

```
slowlog-log-slower-than 10000
```

This indicates if there is an execution time longer than the time in microseconds, in this case one second, it will be logged. The slowlog can be disabled using a value of -1. It can also be set to log every command with a value of 0.

```
slowlog-max-len 128
```

This sets the length of the slow log. When a new command is logged the oldest one is removed from the queue.

These values can also be changed at runtime using the `CONFIG SET` command.

You can view the current length of the slow log using the `LEN` subcommand:

```
redis> slowlog len
(integer) 11
```

Entries can be pulled off of the slow log using the `GET` subcommand.

```
redis> slowlog get 2
1) 1) (integer) 10
   2) (integer) 1616372606
   3) (integer) 600406
   4) 1) "debug"
      2) "sleep"
      3) ".6"
   5) "172.17.0.1:60546"
   6) ""
2) 1) (integer) 9
   2) (integer) 1616372602
   3) (integer) 600565
   4) 1) "debug"
      2) "sleep"
      3) ".6"
   5) "172.17.0.1:60546"
   6) ""
```

The slow log can be reset using the `RESET` subcommand.

```
redis> slowlog reset
OK
redis> slowlog len
(integer) 0
```

Scanning keys

There are a few options that can be passed to redis-cli that will trigger a keyspace analysis. They use the SCAN command so they should be safe to run without impacting operations. You can see in the output of all of them there is a throttling option if needed.

Big Keys:

This option will scan the dataset for big keys and provide information about them.

```
redis-cli --bigkeys

# Scanning the entire keyspace to find biggest
keys as well as
# average sizes per key type. You can use -i 0.1
to sleep 0.1 sec
# per 100 SCAN commands (not usually needed).

[00.00%] Biggest string found so far
'"counter:__rand_int__"' with 6 bytes
[00.00%] Biggest hash  found so far '"myhash"'
with 1 fields
[00.00%] Biggest list  found so far '"mylist"'
with 200000 items

-------- summary -------

Sampled 4 keys in the keyspace!
Total key length in bytes is 48 (avg len 12.00)

Biggest  list found '"mylist"' has 200000 items
Biggest  hash found '"myhash"' has 1 fields
Biggest string found '"counter:__rand_int__"' has
6 bytes

1 lists with 200000 items (25.00% of keys, avg
size 200000.00)
1 hashs with 1 fields (25.00% of keys, avg size
1.00)
2 strings with 9 bytes (50.00% of keys, avg size
4.50)
0 streams with 0 entries (00.00% of keys, avg size
0.00)
0 sets with 0 members (00.00% of keys, avg size
0.00)
0 zsets with 0 members (00.00% of keys, avg size
0.00)
```

Mem Keys:

Similarly to big keys mem keys will look for the biggest keys but also report on the average sizes.

```
redis-cli --memkeys

# Scanning the entire keyspace to find biggest
keys as well as
# average sizes per key type. You can use -i 0.1
to sleep 0.1 sec
# per 100 SCAN commands (not usually needed).

[00.00%] Biggest string found so far
'"counter:__rand_int__"' with 62 bytes
[00.00%] Biggest string found so far
'"key:__rand_int__"' with 63 bytes
[00.00%] Biggest hash  found so far '"myhash"'
with 86 bytes
[00.00%] Biggest list  found so far '"mylist"'
with 860473 bytes

-------- summary -------

Sampled 4 keys in the keyspace!
Total key length in bytes is 48 (avg len 12.00)

Biggest  list found '"mylist"' has 860473 bytes
Biggest  hash found '"myhash"' has 86 bytes
Biggest string found '"key:__rand_int__"' has 63
bytes

1 lists with 860473 bytes (25.00% of keys, avg
size 860473.00)
1 hashs with 86 bytes (25.00% of keys, avg size
86.00)
2 strings with 125 bytes (50.00% of keys, avg size
62.50)
0 streams with 0 bytes (00.00% of keys, avg size
0.00)
0 sets with 0 bytes (00.00% of keys, avg size
0.00)
```

```
0 zsets with 0 bytes (00.00% of keys, avg size
0.00)
```

Hot Keys:

The hot keys scan is only available when the `maxmemory-policy` is set to `volatile-lfu` or `allkeys-lfu`. If you need to identity hot keys you can add this argument to redis-cli.

```
# redis-cli --hotkeys

# Scanning the entire keyspace to find hot keys as
well as
# average sizes per key type. You can use -i 0.1
to sleep 0.1 sec
# per 100 SCAN commands (not usually needed).

[00.00%] Hot key '"key:__rand_int__"' found so far
with counter 37

-------- summary -------

Sampled 5 keys in the keyspace!
hot key found with counter: 37 keyname:
"key:__rand_int__"
```

Monitor

The `MONITOR` command allows you to see a stream of every command running against your Redis instance.

```
redis> > monitor
OK
1616541192.039933 [0 127.0.0.1:57070] "PING"
1616541276.052331 [0 127.0.0.1:57098] "set"
"user:2398423hu" "KutMo"
```

Note: because MONITOR streams back all commands, its use comes at a cost. It has been known to reduce performance by up to 50% so use with caution!

Setting up and using the Redis Log File

The Redis log file is the other important log you need to be aware of. It contains useful information for troubleshooting configuration and deployment errors. If you don't configure Redis logging, troubleshooting will be significantly harder.

Redis has four logging levels, which you can configure directly in `redis.conf` file.

Log Levels:

1. `WARNING`
2. `NOTICE`
3. `VERBOSE`
4. `DEBUG`

Redis also supports sending the log files to a remote logging server through the use of syslog.

Remote logging is important to many security professionals. These remote logging servers are frequently used to monitor security events and manage incidents. These centralized log servers perform three common functions: ensure the integrity of your log files, ensure that logs are retained for a specific period of time, and to correlate logs against other system logs to discover potential attacks on your infrastructure.

Let's set up logging on our Redis deployment. First we'll open our `redis.conf` file

```
$ sudo vi /etc/redis/redis.conf
```

The `redis.conf` file has an entire section dedicated to logging.

First, find the logfile directive in the `redis.conf` file. This will allow you to define the logging directory. For this example lets use `/var/log/redis/redis.log`.

If you'd like to use a remote logging server, then you'll need to uncomment the lines `syslog-enabled, syslog-ident and syslog-facility`, and ensure that `syslog-enabled` is set to yes.

Next, we'll restart the Redis server.

You should see the log events indicating that Redis is starting.

```
$ sudo tail -f /var/log/redis/redis.log
```

And next let's check that we are properly writing to syslog. You should see these same logs.

```
$ less /var/log/syslog | grep redis
```

Finally, you'll need to send your logs to your remote logging server to ensure your logs will be backed up to this server. To do this, you'll also have to modify the rsyslog configuration. This configuration varies depending on your remote logging server provider.

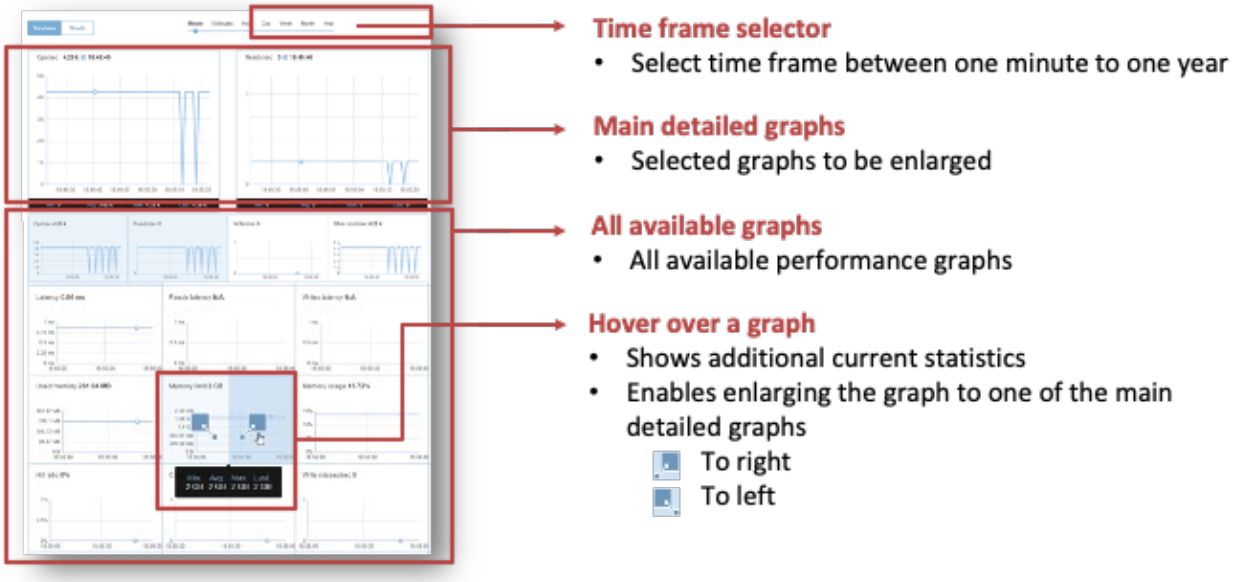
6.5 Observability in Redis Enterprise

You may recall from other units we have shown that Redis Enterprise cluster is a group of resources that Redis databases (even clustered databases) can be deployed onto. There is a cluster manager that is monitoring the health and availability of these resources along with the actual Redis server processes.

A lot of data being collected and in turn used by the cluster manager to make health related decisions like triggering the failover of a primary shard to a replica shard. It is also exposed to the administrator and users of the system.

Dashboards

There are three types of metrics dashboards (cluster, nodes and databases). Every dashboard has some common properties.

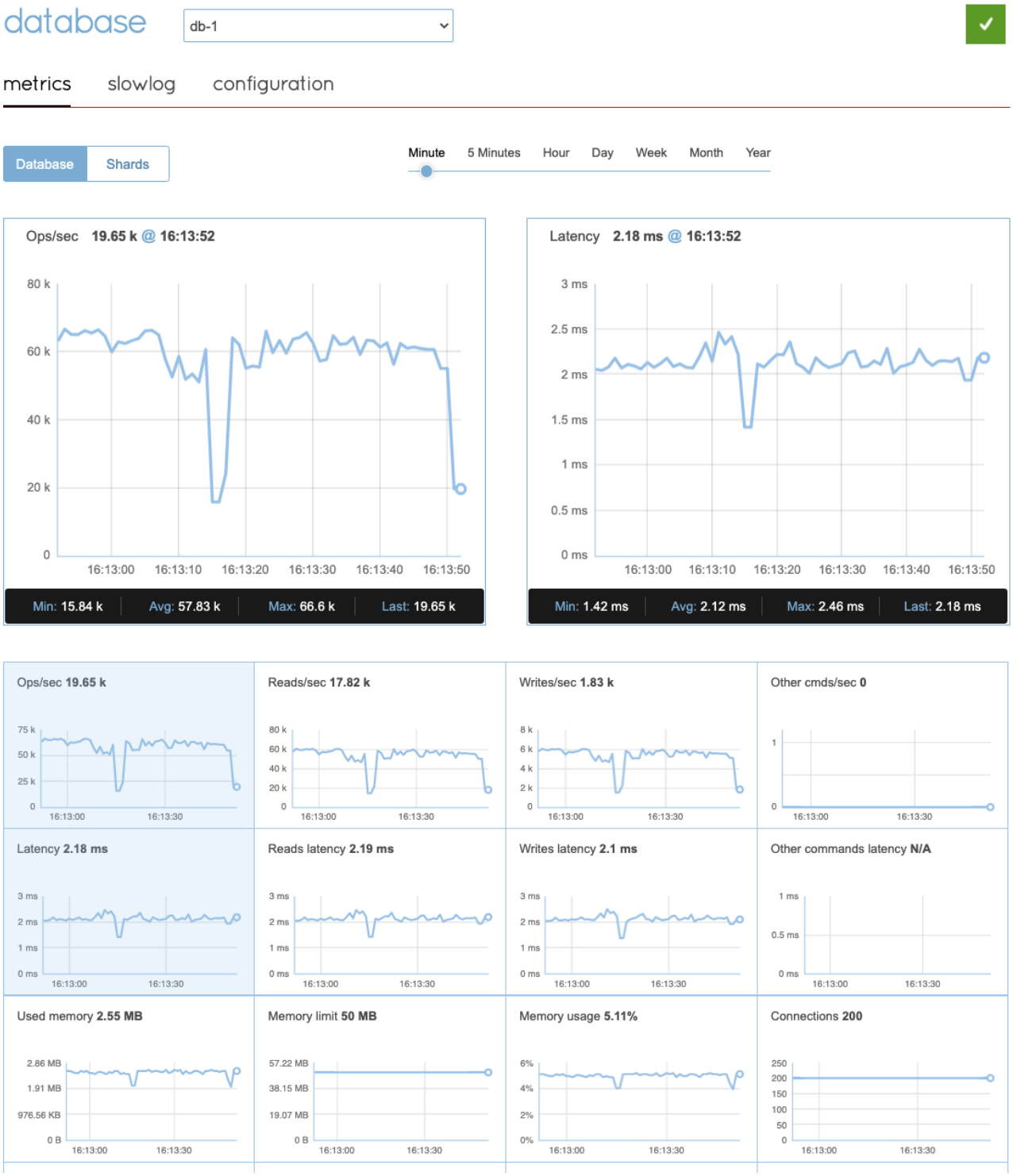


Cluster metrics: aggregated stats for the overall resources of the Redis Enterprise cluster.

Node metrics: stats broken out for each node (machine) in the Redis Enterprise cluster.

Database metrics: stats for each database in the Redis Enterprise cluster.

Example from a database dashboard:



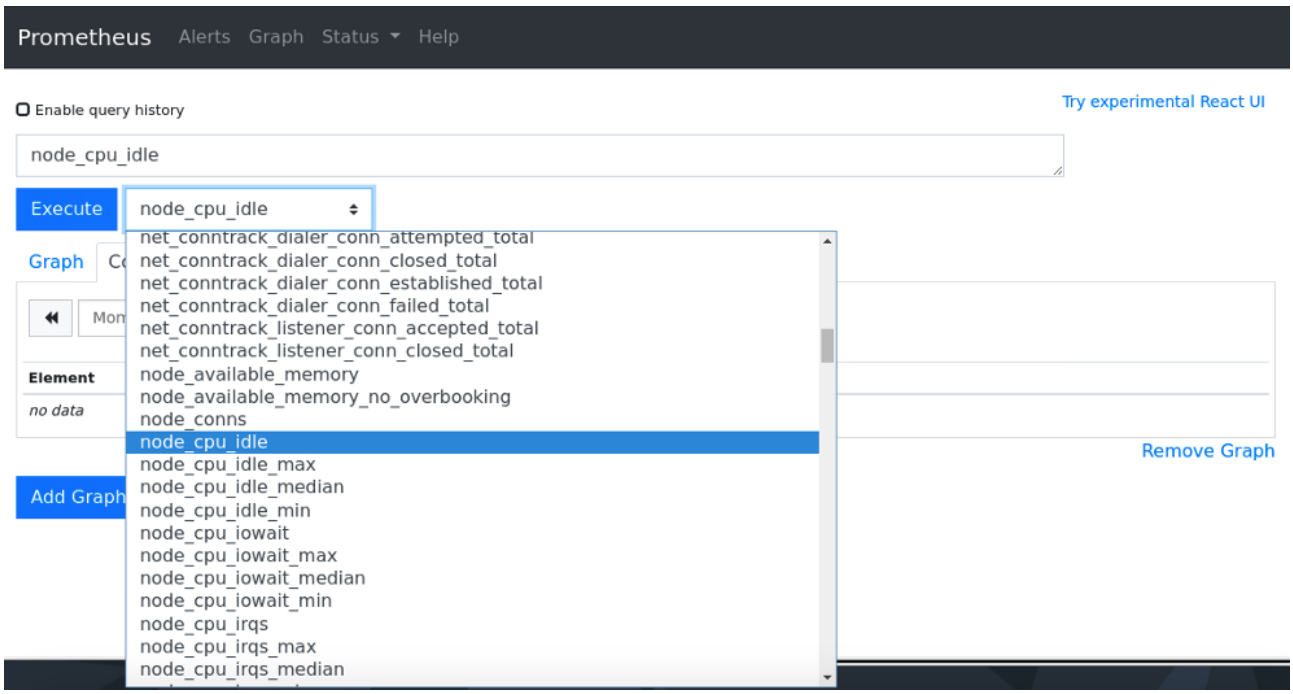
Prometheus metrics endpoint

Prometheus is an open-source systems monitoring and alerting toolkit. Metrics can be exposed to Prometheus using a simple text-based exposition format. Redis Enterprise has a component called the `metrics_exporter` which listens on port `8070` and serves as

a Prometheus scraping endpoint for obtaining metrics in this exposition format.

```
curl -X GET -k https://localhost:8070/
# HELP bdb_total_connections_received
# TYPE bdb_total_connections_received gauge
bdb_total_connections_received{bdb="1",cluster="cl
uster.local"} 0.0
# HELP bdb_shard_cpu_user_max
# TYPE bdb_shard_cpu_user_max gauge
bdb_shard_cpu_user_max{bdb="1",cluster="cluster.lo
cal"} 0.001
# HELP bdb_egress_bytes
# TYPE bdb_egress_bytes gauge
bdb_egress_bytes{bdb="1",cluster="cluster.local"}
0.0
# HELP bdb_read_req_max
# TYPE bdb_read_req_max gauge
bdb_read_req_max{bdb="1",cluster="cluster.local"}
0.0
# HELP bdb_shard_cpu_user
# TYPE bdb_shard_cpu_user gauge
bdb_shard_cpu_user{bdb="1",cluster="cluster.local"
} 0.001
# HELP bdb_read_misses_max
```

A Prometheus server could be set up to pull this data so more complex queries and monitoring could be performed using promQL.



Further visualization could be provided by connecting Grafana to Prometheus. Grafana dashboards can be built for ongoing monitoring.

REST API

Redis Enterprise exposes a REST API that allows for management operations as an alternative to the admin console. Everything that can be done through the admin console (and more) can be accomplished with the REST API. This includes the gathering of metrics and details from the cluster nodes and databases.

```
$ curl -X GET -k -u admin@redis.com:redis123
https://localhost:9443/v1/cluster/stats | json_pp
 % Total    % Received % Xferd Average
Speed  Time    Time     Time Current
                                  Dload Upload  Tota
l  Spent    Left Speed
100 10437 100 10437    0     0  291k      0 --:--:--
--:--:-- --:--:-- 299k
{
  "intervals" : [
     {
        "persistent_storage_avail" : 38071263232,
        "available_flash" : 0,
        "available_memory_no_overbooking" :
14035869992,
        "cpu_system" : 0.01,
        "cpu_idle" : 0.96,
        "free_memory" : 14085783552,
        "ingress_bytes" : 0,
        "available_memory" : 14035865896,
        "cpu_iowait" : 0,
        "interval" : "1sec",
        "stime" : "2021-03-23T23:38:48Z",
        "provisional_flash_no_overbooking" : 0,
        "provisional_memory_no_overbooking" :
11012912876,
        "etime" : "2021-03-23T23:38:49Z",
        "cpu_steal" : 0,
        "conns" : 0,
        "available_flash_no_overbooking" : 0,
```

An interval can also be requested along with a start or end time.

Additional Monitoring Options

The capability to pull metrics through a REST endpoint or to scrape the Prometheus endpoint allows for many third party solutions.

AppDynamics Extension for Redis Enterprise
- Uses REST API metric services
- Fully supported by AppDynamics

New Relic Prometheus OpenMetrics Integration
- Scrapes Redis Enterprise Prometheus endpoint
- Fully supported by New Relic

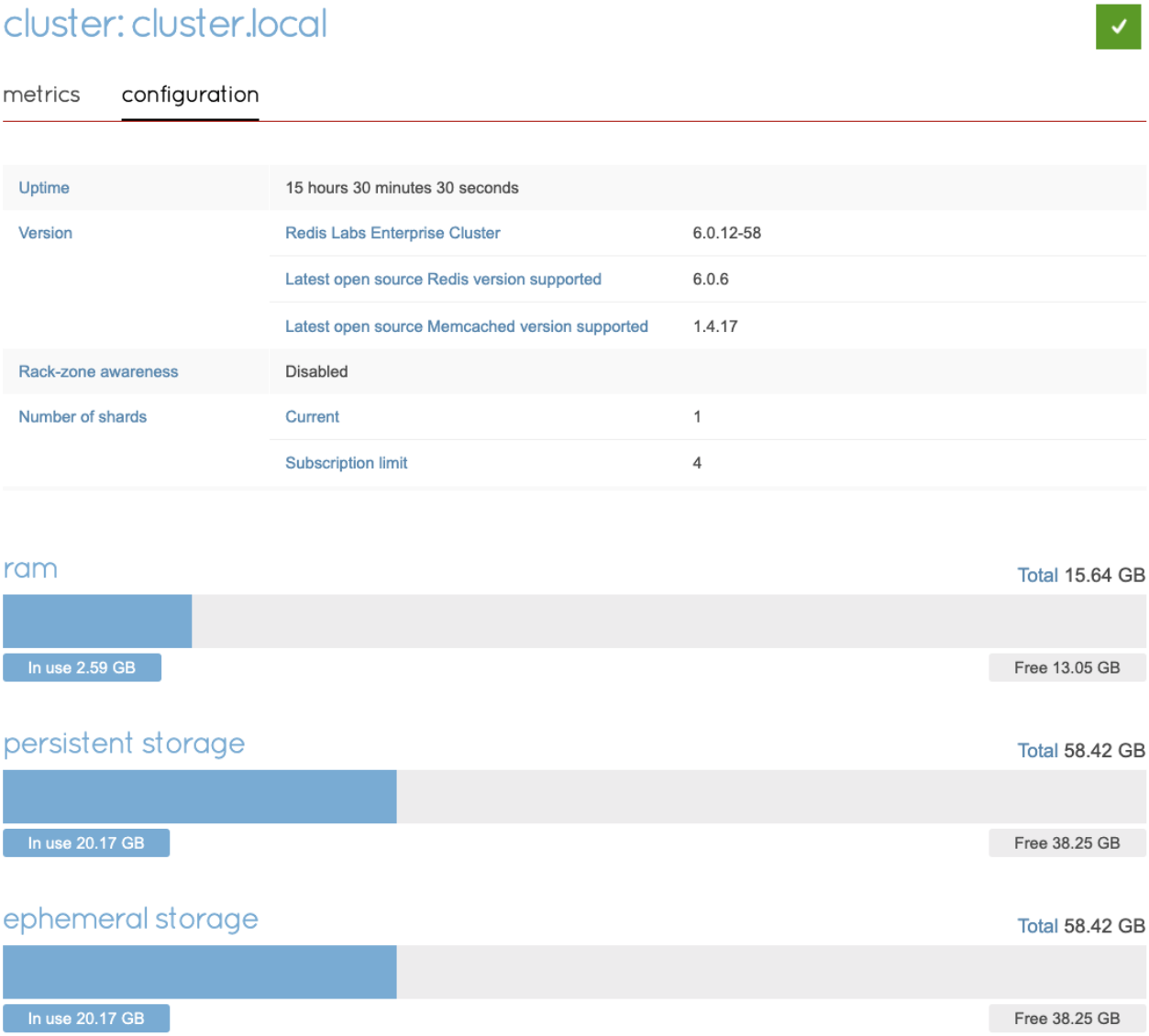
Redis Enterprise Add-on for Splunk
- Uses REST API metric services
- Provided as-is without warranty or support

Pivotal Cloud Foundry Loggregator
- PCF Loggregator scrapes the metrics_exporter and push it to the Firehose

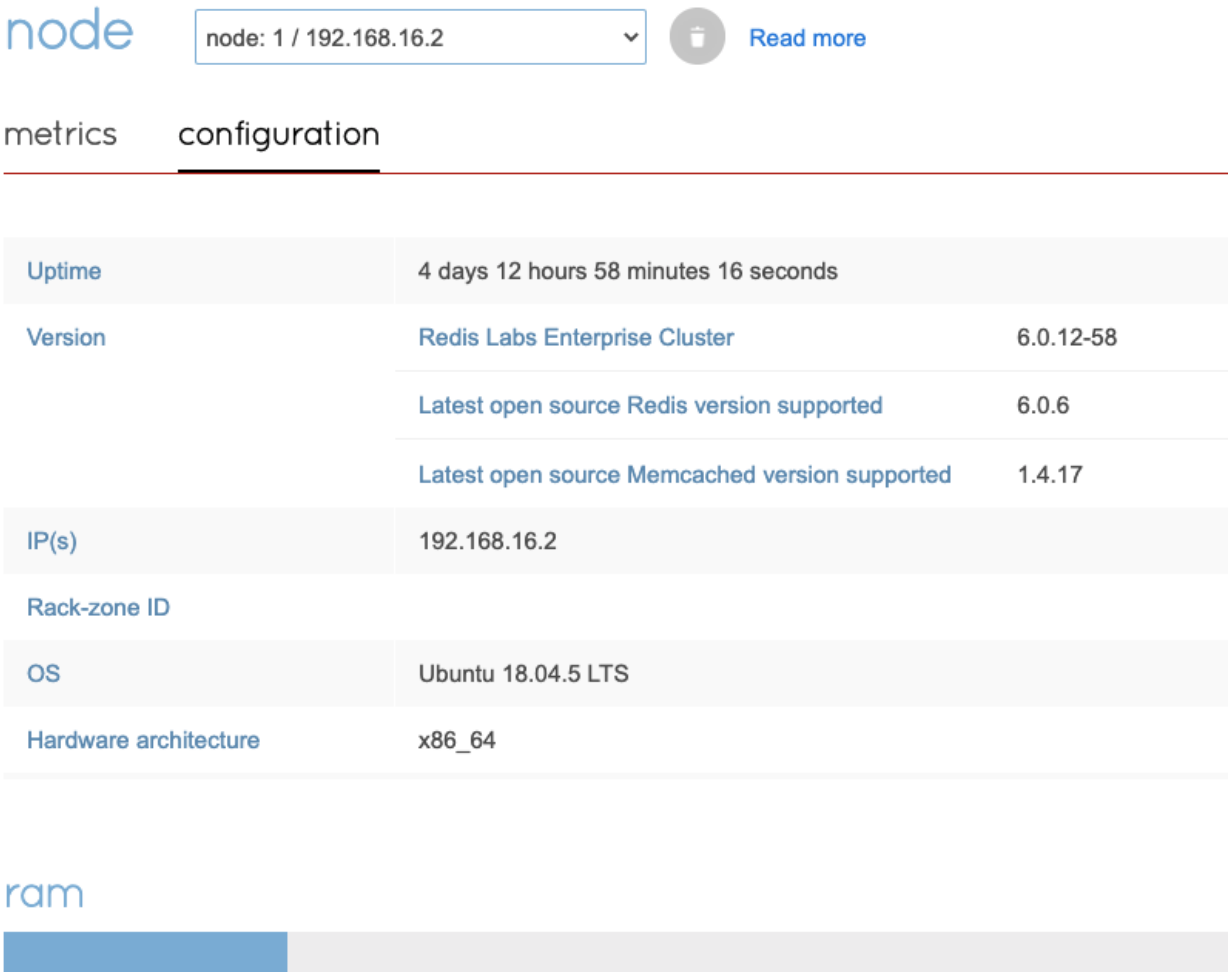
Configuration Data

Each of these same entities (cluster, nodes, databases) has a set of configuration data available for viewing. This can be used to verify versions and settings of the current deployment.

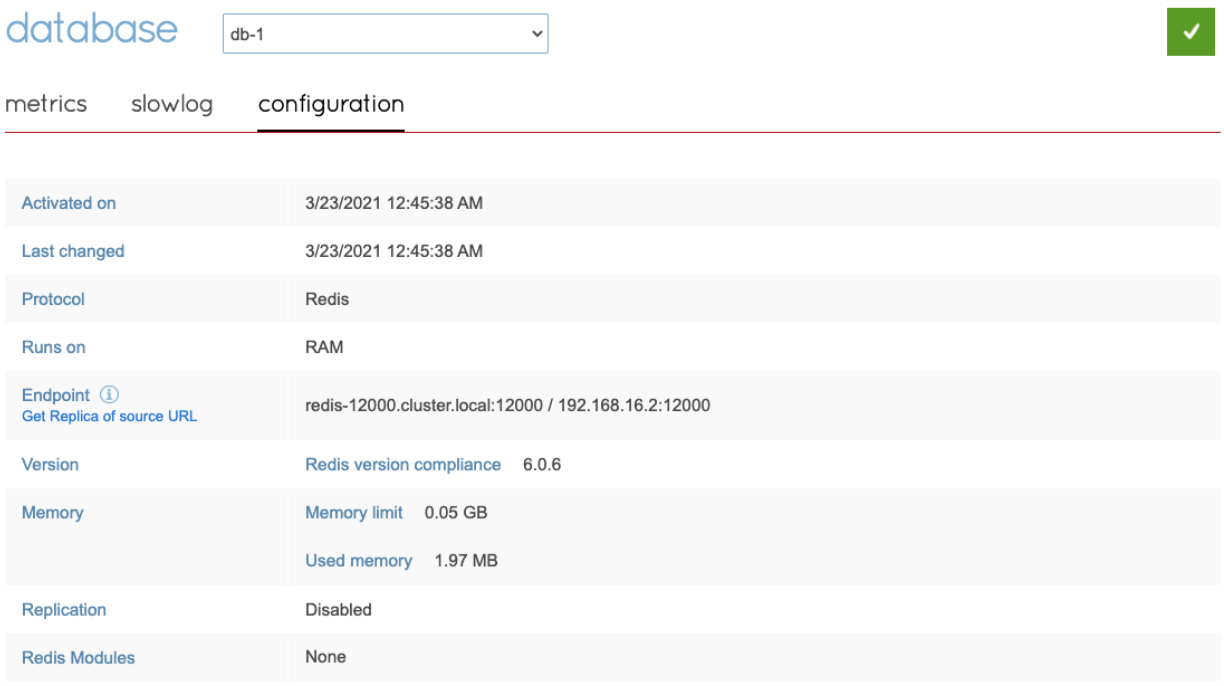
Cluster configuration: Allows you to view the Redis Enterprise version, the Redis server version being used, and other resource details across the cluster.



Nodes configuration: similar to the cluster configuration but node specific. This will also provide the IP address and OS for a given node.



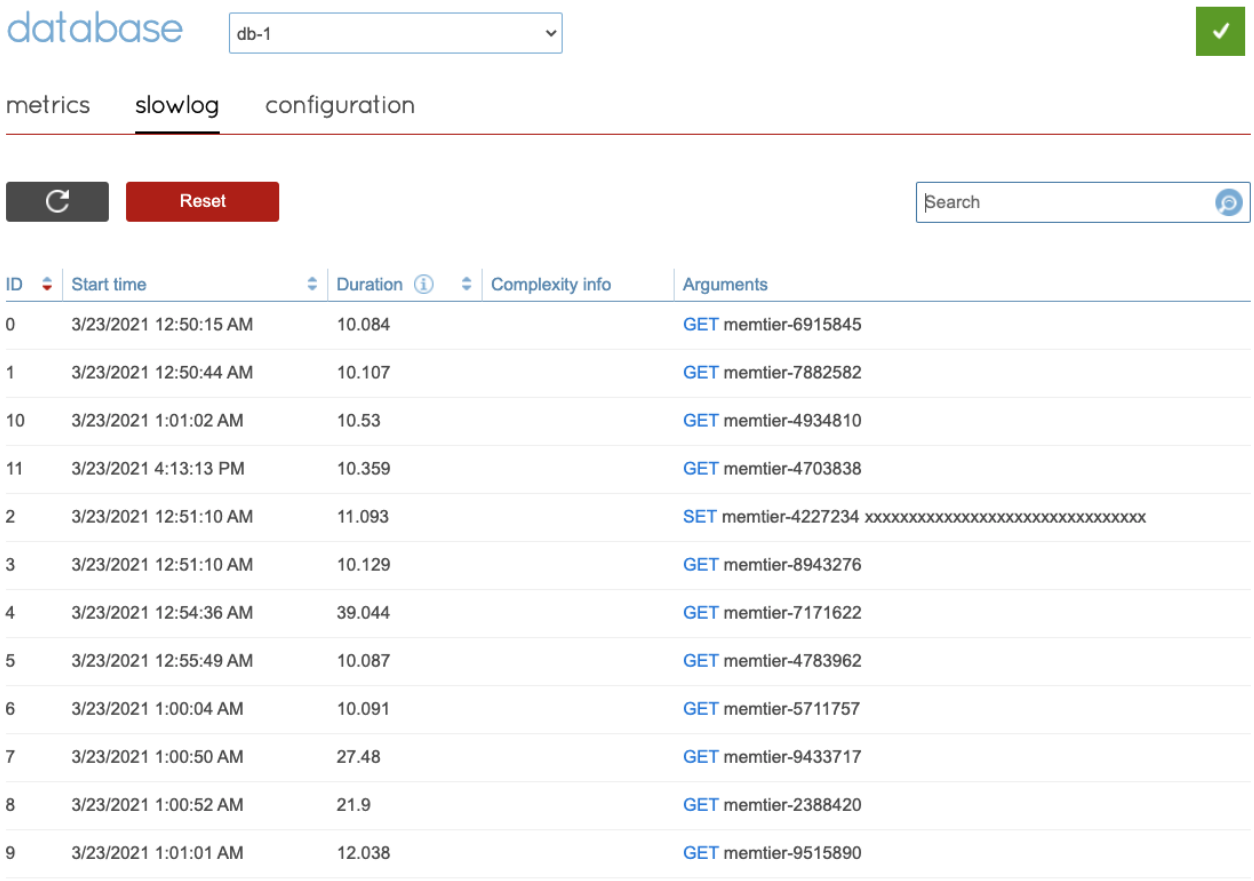
Databases: while the configuration area is often used for updating the database configuration it can be useful for viewing the memory limit, endpoints and other settings for a given database.



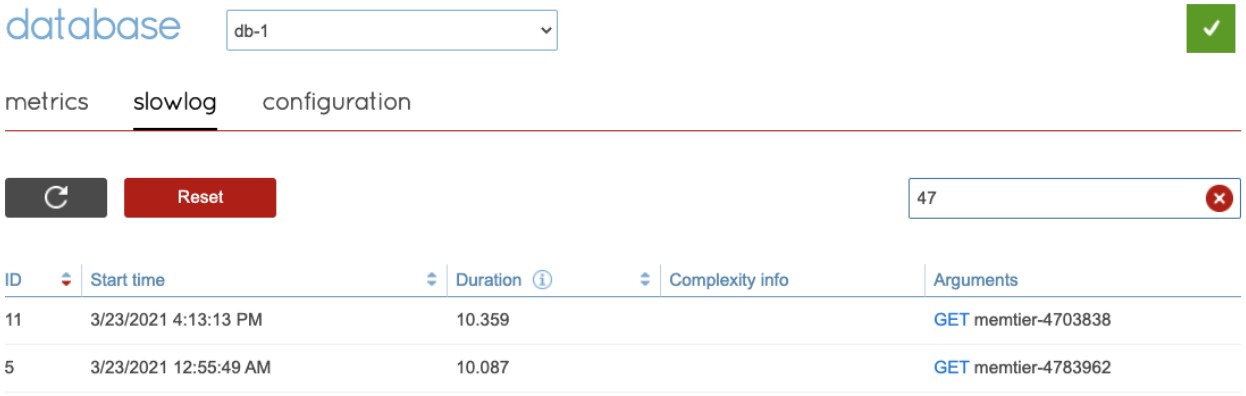
All the configuration data above can be pulled from endpoints in the REST API as well.

Database Slowlog

Each database has a slowlog that is viewable through the admin console. The slowlog can be reset from here as well.



You can search across the slowlog as well:

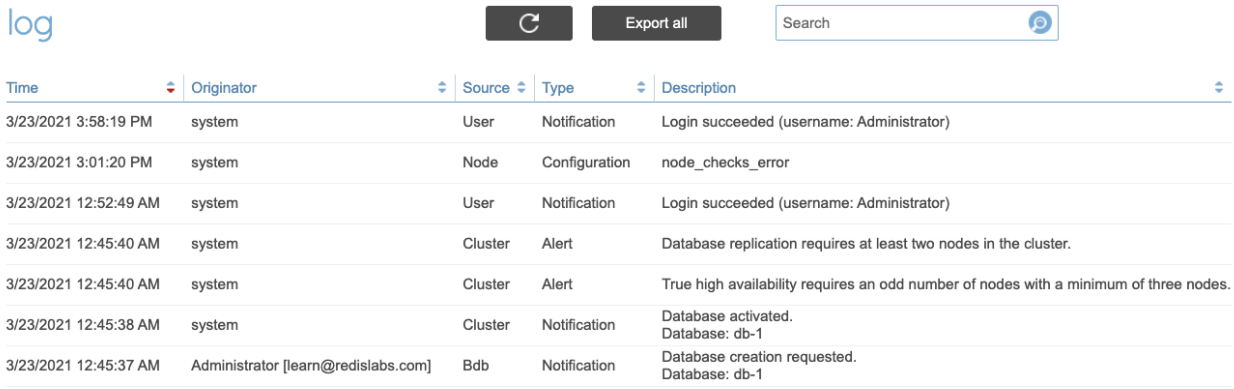


Control Plane Log

There is a log of system and user events on the Redis Enterprise cluster. Each entry includes a timestamp and a description.

Events:
Alerts
Notifications
Configurations

Example:



Cluster and Node Alerts

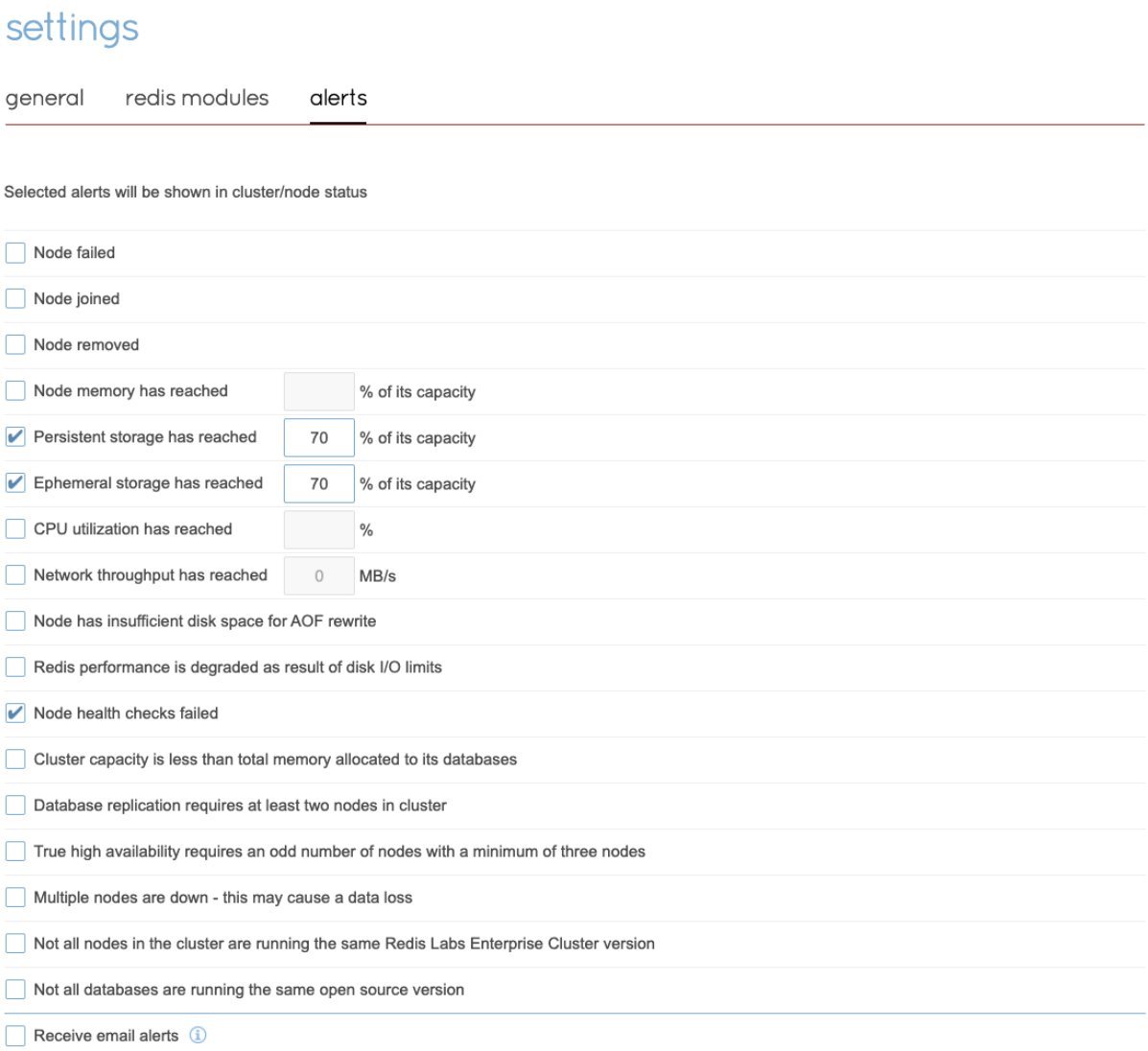
Redis Enterprise has a built-in alerting mechanism. The admin console Alerts page lets you specify which cluster and node level events trigger notifications

Event based alerts can only be turned on or off such as:

- Node failure
- Insufficient space for AOF functionality

Threshold alerts require setting a threshold, such as:

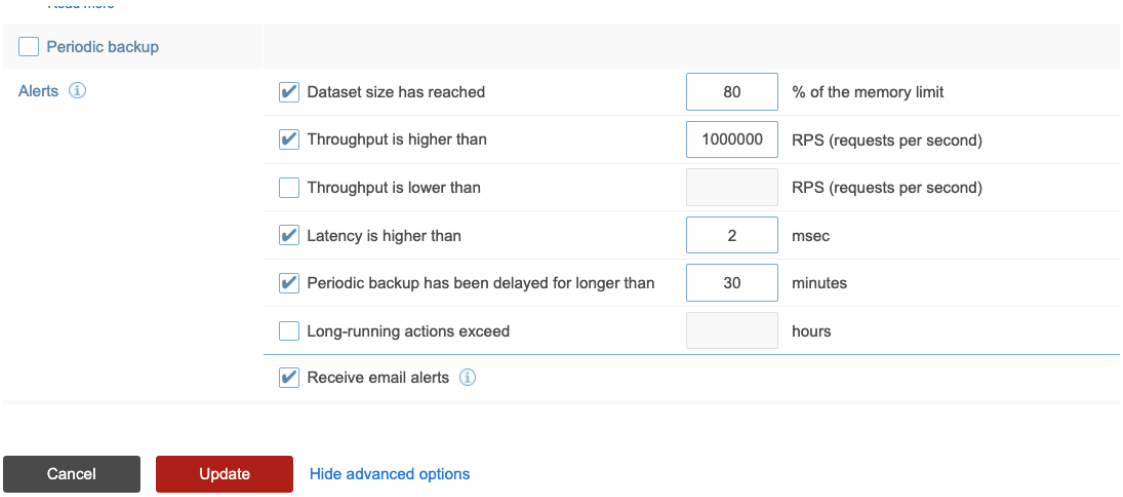
- Node memory has reached a certain percent of capacity
- Storage has reached a certain percent of it's capacity



Database Alerts

Each database has its own Alerts section where you can designate which database events will trigger alert notifications. Some alerts can only be turned on or off, such as Periodic back-up failed while some require setting a threshold, such as Dataset size has reached a certain percentage of its limit.

Raised alerts appear in the database Status field, in the Log page, and can also be sent by email.



Logging

Each Redis Enterprise process has its own log. Here is a list of the processes:

```
$ supervisorctl
alert_mgr                        RUNNING   pid 180,
uptime 0:13:45
ccs                              RUNNING   pid 151,
uptime 0:13:45
cluster_wd                       RUNNING   pid 159,
uptime 0:13:45
cm_server                        RUNNING   pid 181,
uptime 0:13:45
cnm_exec                         RUNNING   pid 174,
uptime 0:13:45
cnm_http                         RUNNING   pid 177,
uptime 0:13:45
crdb_coordinator                 RUNNING   pid 178,
uptime 0:13:45
crdb_worker:crdb_worker_0        RUNNING   pid 172,
uptime 0:13:45
crdb_worker:crdb_worker_1        RUNNING   pid 173,
uptime 0:13:45
dmcproxy                         RUNNING   pid 182,
uptime 0:13:45
```

```
envoy                            RUNNING  pid 161,
uptime 0:13:45
envoy_control_plane              RUNNING  pid 152,
uptime 0:13:45
gossip_envoy                     RUNNING  pid 2026,
uptime 0:12:14
job_scheduler                    RUNNING  pid 200,
uptime 0:13:45
mdns_server                      RUNNING  pid 184,
uptime 0:13:45
metrics_exporter                 RUNNING  pid 192,
uptime 0:13:45
nginx                            RUNNING  pid 2019,
uptime 0:12:15
node_wd                          RUNNING  pid 153,
uptime 0:13:45
pdns_server                      RUNNING  pid 187,
uptime 0:13:45
redis_mgr                        RUNNING  pid 176,
uptime 0:13:45
resource_mgr                     RUNNING  pid 163,
uptime 0:13:45
rl_info_provider                 RUNNING  pid 179,
uptime 0:13:45
saslauthd                        RUNNING  pid 183,
uptime 0:13:45
sentinel_service                 RUNNING  pid 188,
uptime 0:13:45
start_redis_servers              RUNNING  pid 189,
uptime 0:13:45
stats_archiver                   RUNNING  pid 171,
uptime 0:13:45
```

Going over each process and it's logs in detail is out of scope for this course, but we will highlight a few.

`cnm_exec.log:` logs events from the cluster manager state machine

```
INFO sm_event_logger StateMachineHandler0: ===
[bdb:1] STATEMACHINE [SMUpdateBDB] ENDED
SUCCESSFULLY ===
```

dmcproxy.log: captures connection events between an application client and the database

```
ERR no listener on port 12000
```

cluster_wd.log: captures cluster watchdog events such as a node failure

```
INFO: ClusterState: Node Y (xxx.xx.xx.xxx) just
died
```

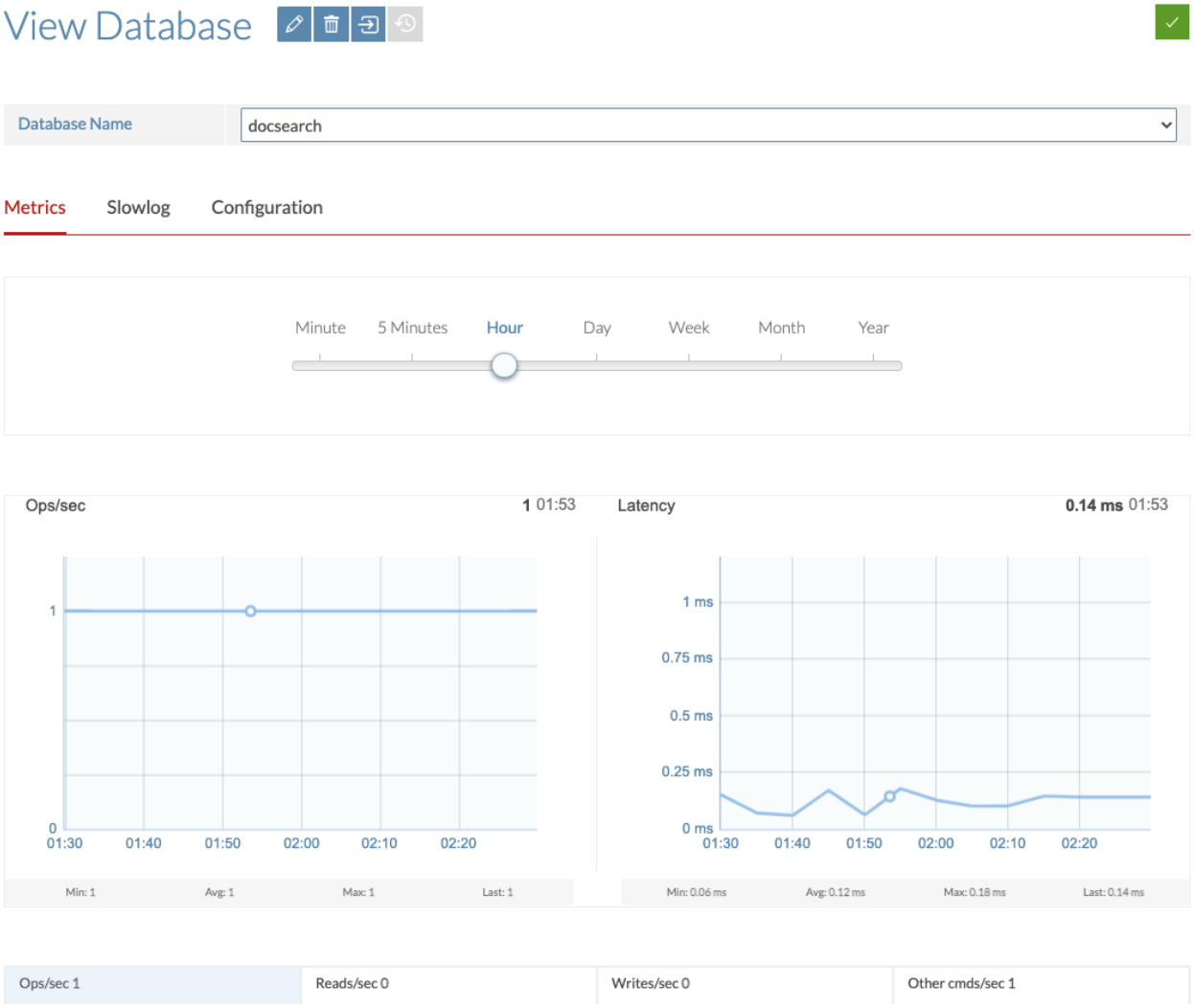
resource_mgr.log: captures state of free memory and shard movement

```
Insufficient memory - moving one shard from node N
to node Y
```

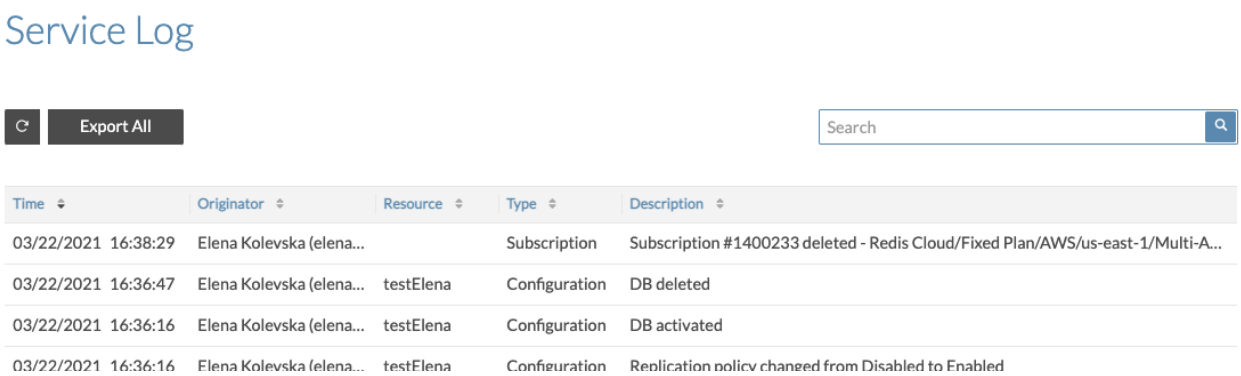
Redis Enterprise Cloud

Redis Enterprise Cloud is a managed service of Redis Enterprise.

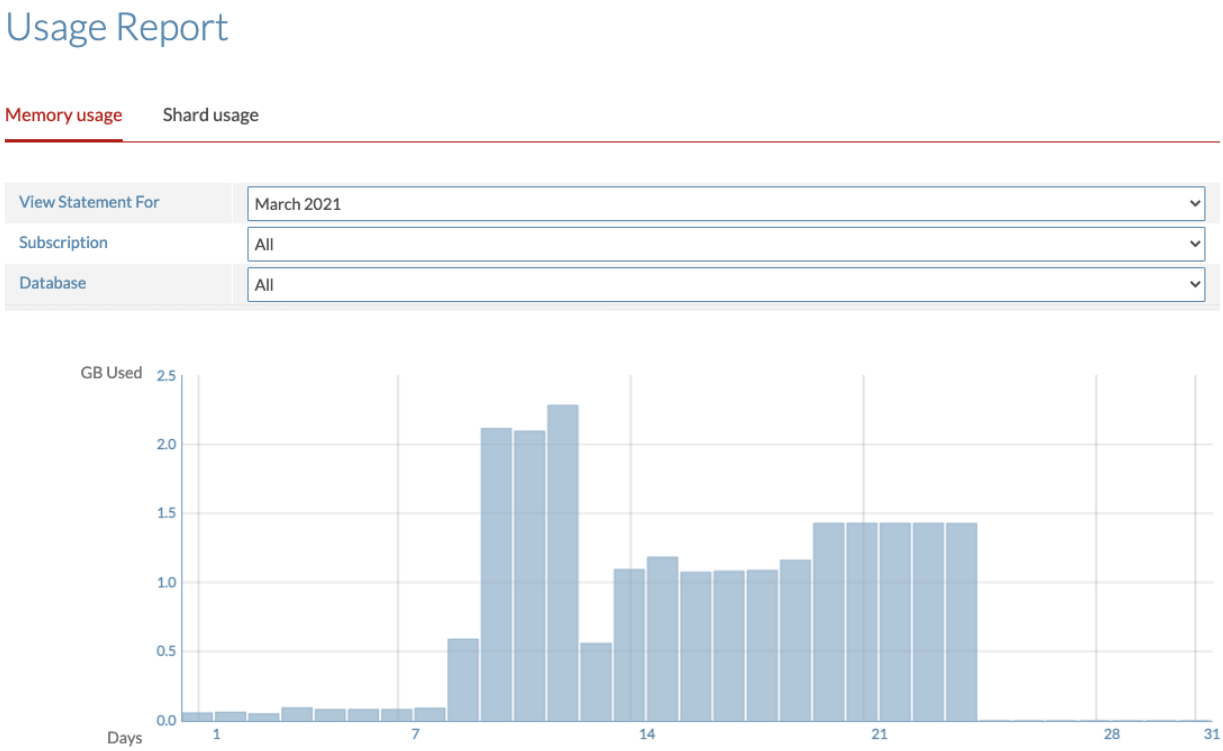
Similar to what we looked at before with Redis Enterprise software, each database has its own metrics dashboard and Slowlog available.



There is an account level service log that provides an event log of account activity.



You can view a month by month usage report for all your subscriptions and databases or drill down into a specific subscription and database.



6.6 Exercise - Redis Enterprise dashboard

Clone this repo if you have not already: https://github.com/redislabs-training/exercises-scaling-redis

Change into the 'observability-monitoring' directory.

Requirements

- docker
- docker-compose
- internet connection

Starting Environment

```
docker-compose up -d
```

Setup

After the `docker-compose -d` command has successfully started up the Redis Enterprise docker container we will need to bootstrap Redis Enterprise. This involves creating a single node cluster. Note that in Redis Enterprise a cluster node equates to an instance resource on which Redis Enterprise is installed and not just where a Redis server process is running.

There is a setup.sh script provided which will get things started so you can focus on the exercises.

Run from this directory:

```
./setup.sh
```

Once that completes you should be able to load Redis Enterprise admin UI in a browser.

https://localhost:8443/

Redis Enterprise creates self-signed certs in the beginning so a secure connection is always created. Since they are not CA signed certificates you will need to allow your browser to accept them.

In the Firefox browser this consists of just clicking through the Advanced button and then find the button to accept and continue. If you are using a Chrome browser you can bypass this warning by just typing 'thisisunsafe' and it will automatically continue through to the site.

Once you bypass the browser warning you should see a login screen where you can use the following credentials to login:

username: admin@redis.com
password: redis123

Once successfully logged in you should see the Redis Enterprise admin console.

Generate some load

The setup script created a database that we can use exposed on port `12000`. Generate some load into the database using another open source load generation utility for Redis called memtier_benchmark. It is included with the Redis Enterprise install so we can simple run:

```
docker-compose exec redis_enterprise_monitoring
memtier_benchmark -p 12000 -x 1000
```

That will just take all the defaults for most of the options except the p param to specify the port which we need to set to `12000` because the database on Redis Enterprise was created to use that port and the x param to indicate how many times to run the default test. This will allow us to be able to see some results in Redis Enterprise.

You can find more information about `memtier_benchmarks` options using the `--help` option.

Cluster Dashboard

Now go back to your browser where you logged in and click the 'cluster' nav menu item. By default this will open the metrics tab.

You should be able to see the overall Cluster Ops/sec and on the right Node 1 Ops/sec. These stats should be the same because in setup (dev/testing only) we have a single node cluster.

Scroll down to the other metrics and hover over 'Free RAM' this should provide a little more detail and give you the option to move this metric to one of the top spots. If you click on the right side box the metric should move to the top right.

Node Dashboard

Navigate to the nodes page by clicking the 'nodes' option in the nav menu. This should just display our single node. If we had joined other nodes to the cluster you would see multiple nodes here.

Click on node 1 to view it's metrics. Now click on the '5 Minutes' text in the time scale to change the sampling time.

You can view the node specific details by clicking on the 'configuration' tab. This will show the version of Redis Enterprise, Redis, IP address, OS and other details.

Database Dashboard

Navigate to the databases page by clicking the 'databases' option in the nav menu. Since a Redis Enterprise cluster can support multi-tenancy you can run multiple databases across the same set of resources. This page will show a list of the databases that have been created in this cluster. The setup script created one database '`db-1`' so click on that now.

You should be taken into the database metrics view immediately. You can see the Ops/sec and the Latency. Scroll down to view other metrics.

What is the used memory at? What is the usage percentage?

How many connections are you getting from the memtier utility?

What are the total keys? How about your hit ratio?

Notice at the top there is a rectangular blue switch box with 'Database' selected. You can click on 'Shards' to view shard specific data.

In our case there won't be much difference in the actual numbers because we are currently running a single shard, but this can be very useful to identify hot shards and get other insights into the database health.

Prometheus Exporter

You can what data is available on the Prometheus exporter endpoint by calling it on port `8070`:

```
docker-compose exec redis_enterprise_monitoring
curl -k https://localhost:8070/metrics
```

REST API

The Redis Enterprise REST API also has a way to pull stats directly. You can try pulling stats for the databases:

```
docker-compose exec redis_enterprise_monitoring
curl -k -u admin@redis.com:redis123 https://
localhost:9443/v1/bdbs/stats
```

You can explore the following query params:

- `interval` – Optional time interval for for which we want stats: 1sec/10sec/5min/15min/1hour/12hour/1week
- `stime` – Optional start time from which we want the stats. Should comply with the ISO_8601 format
- `etime` – Optional end time after which we don't want the stats. Should comply with the ISO_8601 format

Add an interval query param to set the interval time to five minutes.

Stats can be pulled from different endpoints as well:

- `/v1/bdbs/stats - databases`
- `/v1/nodes/stats - nodes stats`
- `/v1/cluster/stats - cluster`

Try each of these endpoints.

Also, the id of a specific entity can be passed, for example if we wanted to just get the stats of our database with the id=1 and not the full array of nodes we could run:

```
docker-compose exec redis_enterprise_monitoring
curl -k -u admin@redis.com:redis123 https://
localhost:9443/v1/bdbs/stats/1
```

Now do the same thing for the nodes endpoint and just get the stats for node 1.

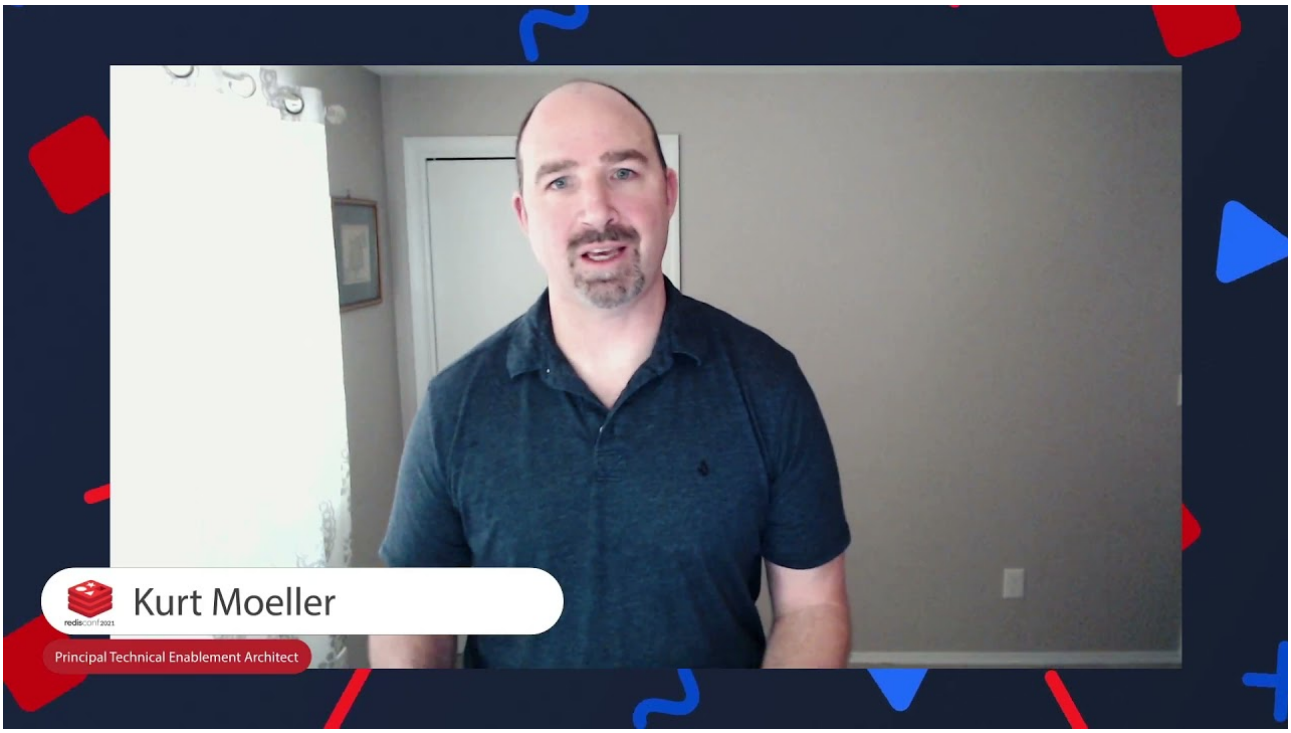
Stop Load Generation

You can just use `ctrl-c` in the terminal window where `memtier_benchmark` is running to to exit.

Stopping the Environment

```
docker-compose down
```

Unit 7: Course Wrap-up



YouTube URL

We covered a lot of material today! You learned how Redis handles requests internally, how to choose and tune a client library, how to make your data durable and all about scalability and high availability. Hopefully this course has helped answer some of those difficult questions when deploying Redis.

- Like how to secure data using TLS and access controls.
- How to avoid downtime even during failures of systems, zones and even regions.
- How to maximize my resources and performance automatically with Redis Enterprise.
- And how to improve the visibility into my Redis deployment with monitoring and observability.

Congrats on finishing the course and hope to see you again soon!