# Getting Started with Redis

## What is Redis?

**Redis** (Remote Dictionary Server) is an **in-memory data store** used as a:

* **Cache**
* **Message broker**
* **Database**

It's known for **speed** due to storing everything in memory.

## Why Use Redis

* 🧱 Rich data types (strings, lists, sets, hashes, etc.)
* 🔄 Pub/Sub messaging
* 📊 Real-time analytics, leaderboard, queues
* 💾 Optional persistence

## Installation

1. docker run --name redis -p 6379:6379 redis

This runs Redis locally on port 6379.

## Data Types

* **String**: Simple key-value (e.g. name), SET key value, GET key
* **List**: Queue or Stack, LPUSH mylist "a", RPUSH, LPOP, LRANGE mylist 0 -1
* **Set**: Unique items (e.g. tags), SADD myset "a", SMEMBERS myset, SREM
* **Hash**: Object-like (e.g. user info), HSET user name "Ali", HGET user name, HGETALL user
* **Sorted Set**: Leaderboards (score-based), ZADD scores 100 "Ali", ZRANGE scores 0 -1 WITHSCORES

### Number

1. SET count 1

2. INCR count # count becomes 2

3. INCRBY count 10 # count becomes 12

4. DECR count # count becomes 11

### String

1. SET message "Hello"

2. APPEND message ", World!"

3. GET message # "Hello, World!"

4. STRLEN message # returns length of the value

### List

1. LPUSH tasks "Task 1” #Push to the left

2. LPUSH tasks "Task 2"

3. LPUSH tasks "Task 3"

5. LRANGE tasks 0 -1 # Get all elements

1. RPUSH queue "user1"

2. RPUSH queue "user2"

3. RPUSH queue "user3"

4. LRANGE queue 0 -1

5. LPOP queue # Pops from the left

6. RPOP queue # Pops from the right

7. LLEN tasks # Get number of elements

8. LINDEX tasks 0 # Get first item

9. LSET tasks 1 "New Task" # Replace value at index

10. LREM tasks 1 "Task 2" # Remove 1 occurrence of "Task 2"

11. LTRIM tasks 0 1 # Keep only first 2 items (index 0 & 1)

10. BLPOP queue 0 # Wait until an item is available

Think of RPUSH + LPOP = **FIFO Queue**

### Practice

1. LPUSH fruits "apple"

2. LPUSH fruits "banana"

3. RPUSH fruits "cherry"

4. LRANGE fruits 0 -1

5.

6. LPOP fruits

7. RPUSH fruits "mango"

8. LRANGE fruits 0 -1

9.

10. LLEN fruits

11. LINDEX fruits 0

12. LTRIM fruits 0 1

Applications of List (Queues and Stack)

* Job Queues (e.g., background tasks)
* Chat Messages (recent messages in a room)
* Undo History (stack behavior)

### Redis **Sets**

Redis **Sets** are collections of **unique strings**. They’re **unordered**, meaning the order of items is not guaranteed — but duplicates are **automatically removed**.

Perfect for:

* Tags
* User roles
* "Liked by" features
* Recommendation systems

1. SADD colors "red"

2. SADD colors "blue" "green" "yellow"

3. SADD colors "red" # duplicate, will be ignored

4.

5. SMEMBERS colors # View all items

Output: Unordered unique list: ["blue", "red", "green", "yellow"]

1. SREM colors "yellow"

2. SMEMBERS colors

1. SISMEMBER colors "green" # Returns 1 if exists, 0 otherwise

2. SCARD colors # Count of elements

3. SRANDMEMBER colors # Get a random member

4. SPOP colors # Remove and return a random member

5.

Set Operations: INTERSECTION / UNION / DIFFERENCE

1. SADD set1 "a" "b" "c"

2. SADD set2 "b" "c" "d"

3. SINTER set1 set2 # ["b", "c"]

4. SUNION set1 set2 # ["a", "b", "c", "d"]

5. SDIFF set1 set2 # ["a"]

Redis Hashes – Storing Objects (Key-Value Inside a Key)

1. HSET user:1 name "Ali" age "25" email "ali@example.com"

2. HGET user:1 name

3. HGETALL user:1

4. HMGET user:1 name email

5. HMSET user:2 name "Sara" age "22" email [sara@example.com](mailto:sara@example.com)

### HSET

HMSET is technically deprecated but still widely used. HSET with multiple fields is recommended in newer versions.

1. HSET user:1 age "26"

2. HDEL user:1 email

3. HKEYS user:1 # All field names

4. HVALS user:1 # All field values

5. HEXISTS user:1 name # 1 if exists, 0 otherwise

6. HLEN user:1

7. HINCRBY user:1 score 10

8. HINCRBY user:1 score -5

9.

Practice

1. HSET book:101 title "Redis Basics" author "Ali" pages "120"

2. HGET book:101 title

3. HGETALL book:101

4. HINCRBY book:101 pages 30

5. HDEL book:101 author

6.

### Sorted Set

A **Sorted Set** in Redis is like a regular set, but every item has an associated **score** that determines its order.

This makes it perfect for:

* **Leaderboards**
* **Ranking systems**
* **Priority queues**
* **Time-based sorting**

1. ZADD leaderboard 100 "Ali"

2. ZADD leaderboard 200 "Sara"

3. ZADD leaderboard 150 "Usman"

4.

5. // View Items Ordered by Score (Low → High)

6. ZRANGE leaderboard 0 -1 WITHSCORES

7.

8. // View Items Ordered by Score (High → Low)

9. ZREVRANGE leaderboard 0 -1 WITHSCORES

10.

11. // From lowest score to highest:

12. ZRANK leaderboard "Ali"

13.

14. // From highest to lowest:

15. ZREVRANK leaderboard "Ali"

16.

17. // Update Score (Increment)

18. ZINCRBY leaderboard 50 "Ali"

19. // Now Ali’s score is 100 + 50 = 150

20.

21. // Remove a Member

22. ZREM leaderboard "Sara"

23.

1. // Get Members within Score Range

2. ZRANGEBYSCORE leaderboard 100 200

3.

4. // Remove Members by Score

5. ZREMRANGEBYSCORE leaderboard 0 100

6.

7. // Count Members in Score Range

8. ZCOUNT leaderboard 100 200

9.

Practice

1. ZADD game:scores 500 "Player1"

2. ZADD game:scores 300 "Player2"

3. ZADD game:scores 450 "Player3"

4.

5. ZRANGE game:scores 0 -1 WITHSCORES

6. ZREVRANGE game:scores 0 -1 WITHSCORES

7.

8. ZINCRBY game:scores 100 "Player2"

9. ZREVRANK game:scores "Player2"

10.

**Use Cases for Sorted Sets**

| **Use Case** | **Example** |
| --- | --- |
| Leaderboards | Game scores, user points |
| Time-ordered items | Use timestamps as scores |
| Rate limiting | Track access time per user |
| Feed ranking | Posts ordered by score/relevance |

### Redis Pub/Sub – Publish and Subscribe

Redis Pub/Sub (Publish/Subscribe) allows messages to be sent and received in real-time between multiple clients.

It's used for:

* Real-time chat apps
* Notifications
* Live dashboards
* Event broadcasting between microservices

SUBSCRIBE news

// This client is now listening for messages on the news channel.

In another terminal:

PUBLISH news "Breaking: Redis 7 released!"

The subscriber will receive:

1) "message"

2) "news"

3) "Breaking: Redis 7 released!"

// Subscribe to Multiple Channels

SUBSCRIBE news sports weather

// Pattern-based Subscriptions

PSUBSCRIBE tech.\*

// PSUBSCRIBE allows you to match channel names with patterns.

PUBLISH tech.redis "New Redis Tutorial Available"

// Unsubscribe

UNSUBSCRIBE news

PUNSUBSCRIBE tech.\*

**Behind the Scenes**

* **Subscribers** are passive. They can only **receive** messages.
* **Publishers** can **broadcast** to channels.
* Redis does **not store** messages. If nobody is subscribed, the message is lost.

|  |  |
| --- | --- |
| **Scenario** | **Example** |
| Real-time Chat | Channels for each chat room |
| Notification System | Push alerts to connected clients |
| Microservice Events | Broadcast updates between services |
| Live UI Updates | Real-time dashboards or price tickers |

## Redis Persistence – Making Data Durable

* **RDB (Snapshotting)**: Saves the DB at intervals.
* **AOF (Append Only File)**: Logs every write operation.
* You can enable both for durability.

By default, Redis is an **in-memory database**, which means that once the server restarts, the data is lost. However, Redis offers ways to **persist data** to disk to make it durable across restarts.

Redis provides two main persistence options:

### RDB (Redis Database) Persistence

* RDB snapshots are point-in-time snapshots of your dataset.
* It’s fast and suitable for backups.

**How to Enable RDB:**

You can configure RDB persistence in the redis.conf file with the SAVE directive:

# Save every 60 seconds if at least 1000 keys have changed

SAVE 60 1000

Redis will take a snapshot of your dataset based on these conditions.

**Commands:**

* **BGSAVE**: Initiates a **background** save (non-blocking).
* **LASTSAVE**: Returns the timestamp of the last RDB save.

### AOF (Append-Only File) Persistence

* **AOF** logs every write operation received by the server.
* It provides **greater durability** than RDB but is slower because every write is logged to disk.

In redis.conf, enable AOF:

appendonly yes

appendfsync everysec # Write to disk every second (recommended)

* everysec (default): **Write every second** (a good balance between performance and durability).
* always: Writes after every command (slow, but safest).
* no: **Disables AOF**.

BGREWRITEAOF: Rewrites the AOF file in the background to make it more compact.

## Expiry & TTL

1. SET session\_id "abc123" EX 300 # Set with 5 min expiry

2. TTL session\_id # Check time-to-live

## Conditional SET (NX / XX)

You can **only set a value if** the key doesn't exist (NX) or only if it does (XX):

SET token "xyz" NX # Will only set if token doesn't exist

SET token "abc" XX # Will only set if token exists

Useful in:

* Preventing overwrites
* Distributed locking

## Transactions

1. MULTI

2. SET name "Ali"

3. SET age 25

4. EXEC

## Redis with Node.js

``` npm install redis```

```

import { createClient } from 'redis';

const client = createClient({

port: ‘6379’,

host: ‘’

});

await client.connect();

await client.set('name', 'Ali');

const value = await client.get('name');

console.log(value);

```

## Redis Use Cases

* **Caching layer** (e.g., database query caching)
* **Rate limiting** (track requests per IP)
* **Session storage** (e.g., user auth sessions)
* **Leaderboards** (with Sorted Sets)
* **Job Queues** (via List or libraries like Bull)

### Rate Limiting Example (Per IP)

1. INCR ip:192.168.1.1

2. EXPIRE ip:192.168.1.1 60

## Clustering & Replication

* **Replication**: One master, multiple replicas.
* **Clustering**: Horizontal scaling by sharding data across nodes.
* Use Redis Sentinel for high availability.

### Redis Cluster & Scaling

Redis **Cluster** is designed to distribute data across multiple Redis nodes, enabling horizontal scaling and higher availability.

**1. What is Redis Cluster?**

* Redis Cluster allows you to **partition** your data across **multiple Redis instances**.
* Redis Cluster ensures **data availability** by replicating data on multiple nodes.
* Redis Cluster also automatically handles **node failures** and **rebalancing** of data.

**Cluster Basics:**

* **Slots**: Redis Cluster divides the data into **16,384 slots**. Each key is assigned to a slot.
* Redis instances in the cluster are responsible for a subset of these slots.
* Redis Cluster supports **master-slave replication**, and if a master node fails, its slave can take over.

**How Redis Cluster Works**

* Redis Cluster allows data to be **sharded** across multiple nodes. Each node holds a subset of data.
* Clients can directly interact with a **hash slot** to determine which node stores the data.
* **Failover** happens automatically if a master node goes down, with a slave promoted to master.

**Setting Up Redis Cluster**

Redis Cluster requires at least **3 master nodes** (preferably with **slaves** for replication). The setup involves multiple steps:

**Start Redis Nodes:**

redis-server --port 7000 --cluster-enabled yes --cluster-config-file nodes.conf --cluster-node-timeout 5000

**Create the Cluster:**

After starting the Redis nodes, create the 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

* --cluster-replicas 1: Creates 1 replica for each master node.

**Cluster Operations**

* **SET/GET**: Redis Cluster automatically routes your operations to the correct node.
* **MIGRATE**: Moves data between nodes.
* **CLUSTER NODES**: Shows cluster status and topology.

CLUSTER NODES

* **CLUSTER INFO**: Provides basic info about the cluster's health and state.

CLUSTER INFO

**Cluster Advantages & Considerations**

* **Scalability**: Easily scale horizontally by adding more nodes to the cluster.
* **Fault Tolerance**: Redis Cluster supports automatic failover.
* **Complexity**: Setting up and managing a Redis Cluster is more complex than running a standalone Redis instance.

## Real-World Patterns & Best Practices

Now that you've got the technical foundations, let's explore **best practices** and **real-world patterns** for Redis.

### 1. Cache-aside Pattern

Use Redis as a **cache** to offload frequent database queries:

1. Check Redis cache for data.
2. If **not present**, query the database.
3. **Store** the database result in Redis for future use.

**Example:**

const data = redis.get('user:123') || db.query('SELECT \* FROM users WHERE id=123');

* **Problem**: Redis might get stale data.
* **Solution**: Use **TTL (Time-to-Live)** to ensure cache expires after a certain time.

### 2. Job Queue / Work Queue

Redis is commonly used to implement **job queues** for background processing.

* **Producer**: Adds jobs to the Redis queue.
* **Consumer**: Retrieves and processes jobs from the queue.

Use **Lists** or **Sorted Sets** for this pattern.

### 3. Rate Limiting

Redis is ideal for **rate limiting** with simple **counter** patterns:

1. Track requests using HINCRBY or INCRBY.
2. Set a **TTL** on the key to reset the count after a period.

Example (limit to 100 requests per hour):

INCRBY user:requests 1

EXPIRE user:requests 3600

### 4. Pub/Sub for Event-Driven Systems

Redis Pub/Sub is widely used in **event-driven** systems:

* Use **channels** to broadcast messages.
* Use **patterns** to handle dynamic channels.

### 5. Data Expiry with TTL

Use **TTL (Time to Live)** to automatically expire keys:

SET user:123 "active" EX 3600

This will automatically remove the user:123 key after 1 hour.

### 6. Backup & Recovery

* Use **RDB snapshots** for periodic backups.
* Use **AOF** for a more **durable** approach (but slower).

### 7. Monitoring and Metrics

* Use **Redis INFO** to check server stats.
* Monitor **memory usage** and **slow queries** to avoid issues.

## Security

* Use requirepass in redis.conf to protect Redis.
* Don’t expose Redis to the public internet without protection.
* Enable TLS in production.

## Applications

### Step 1: Installing Redis and Node.js Redis Client

Before we begin, ensure you have **Redis** running and Node.js installed. We will use the popular **ioredis** client to interact with Redis in Node.js.

1. Install Redis Locally

* If you haven't already installed Redis, you can do so by following the official installation guide: Redis Installation.
* To start Redis locally, you can run the following:

redis-server

2. Install ioredis in Node.js

In your Node.js project, install the ioredis package to interact with Redis:

npm install ioredis

### Step 2: Connecting to Redis from Node.js

Let's start by connecting to Redis from Node.js using **ioredis**.

**1. Basic Connection**

Create a new JavaScript file, say index.js, and set up a connection to Redis:

const Redis = require('ioredis');

// Create a new Redis instance

const redis = new Redis();

// Test connection

redis.ping().then((result) => {

console.log(result); // Should print "PONG"

}).catch((err) => {

console.error('Error connecting to Redis:', err);

});

This should connect to a locally running Redis server. If you have a Redis server running remotely, you can pass the connection details like so:

const redis = new Redis({

host: 'your-redis-host',

port: 6379,

password: 'your-password' // If Redis is password-protected

});

### Step 3: Basic Redis Commands in Node.js

Now, let's interact with Redis through **ioredis**.

**1. Set and Get Values**

Set a key in Redis and retrieve it:

redis.set('user:1000', JSON.stringify({ name: 'John Doe', age: 30 }))

.then(() => {

return redis.get('user:1000');

})

.then((result) => {

const user = JSON.parse(result);

console.log(user); // { name: 'John Doe', age: 30 }

})

.catch((err) => console.error('Error:', err));

### 2. Using Expiration (TTL)

Set an expiration time (TTL) for a key:

redis.set('session:abc123', 'active', 'EX', 60)

.then(() => {

console.log('Session set with TTL');

})

.catch((err) => console.error('Error:', err));

This key will expire after 60 seconds.

### **Step 4:** Using Redis as a Cache in Node.js

One of the most common uses of Redis with Node.js is caching. Let's build a simple caching mechanism.

**1. Simple Cache System**

For example, imagine you have an API that fetches user data from a database. Instead of querying the database each time, we can cache the data in Redis.

const getUserFromDatabase = (userId) => {

// Simulate database call (e.g., from MongoDB or SQL)

return new Promise((resolve) => {

setTimeout(() => {

resolve({ id: userId, name: 'John Doe', age: 30 });

}, 2000);

});

};

const getUser = async (userId) => {

const cacheKey = `user:${userId}`;

// Try to fetch from Redis cache

const cachedUser = await redis.get(cacheKey);

if (cachedUser) {

console.log('Cache hit');

return JSON.parse(cachedUser);

}

console.log('Cache miss');

// If not in cache, fetch from database

const user = await getUserFromDatabase(userId);

// Store the result in cache for 5 minutes

redis.set(cacheKey, JSON.stringify(user), 'EX', 300);

return user;

};

// Example usage

getUser(1000).then((user) => {

console.log(user);

});

**2. Caching Benefits**

* **Faster Response**: Redis reduces the time for fetching data by storing results in memory.
* **Reduced Load on Database**: By caching frequently requested data, you reduce the number of database queries.

### **Step 5:** Session Management with Redis in Node**.js**

Redis is also commonly used for **session management** in web applications. You can store user session data in Redis and make it persist across multiple requests.

**1. Install Session Middleware**

If you're using **Express.js** for your Node.js application, you can use express-session with Redis to handle sessions.

npm install express express-session connect-redis

**2. Set up Session with Redis**

Now, configure the session store to use Redis:

const express = require('express');

const session = require('express-session');

const RedisStore = require('connect-redis')(session);

const Redis = require('ioredis');

const app = express();

const redisClient = new Redis();

// Configure session middleware

app.use(session({

store: new RedisStore({ client: redisClient }),

secret: 'your-secret-key',

resave: false,

saveUninitialized: false,

cookie: { secure: false } // Set to `true` if using HTTPS

}));

// Route to set session data

app.get('/set-session', (req, res) => {

req.session.user = { id: 1000, name: 'John Doe' };

res.send('Session data set!');

});

// Route to get session data

app.get('/get-session', (req, res) => {

if (req.session.user) {

res.send(`Hello, ${req.session.user.name}`);

} else {

res.send('No session data found');

}

});

// Start the server

app.listen(3000, () => {

console.log('Server running on http://localhost:3000');

});

**3. Session Expiry**

You can configure session expiry through the **cookie** setting or through Redis expiration:

cookie: { maxAge: 3600000 } // Session will expire in 1 hour

### Step 6: Redis Pub/Sub with Node.js

Another cool feature of Redis is **Pub/Sub** (Publish/Subscribe), which is great for real-time applications like chat or notifications.

**1. Implementing Pub/Sub in Node.js**

**Publisher**:

const Redis = require('ioredis');

const redisPublisher = new Redis();

setInterval(() => {

redisPublisher.publish('chat:room1', 'Hello from Redis!');

}, 5000);

**Subscriber**:

const redisSubscriber = new Redis();

redisSubscriber.subscribe('chat:room1', (err, count) => {

if (err) {

console.error('Failed to subscribe to channel');

} else {

console.log(`Subscribed to ${count} channels`);

}

});

redisSubscriber.on('message', (channel, message) => {

console.log(`Received message on ${channel}: ${message}`);

});

**🔹 2. Real-time Messaging**

In a real-world scenario, the publisher can be any event source (e.g., a new chat message or system notification), and the subscriber can be the clients listening for updates.

### Step 7: Real-World Application Example

Here’s a high-level structure of a **real-world application** using Redis with Node.js:

1. **User Authentication**: Store and validate sessions with Redis.
2. **Caching**: Cache frequently accessed data (e.g., user profiles, product listings) to reduce database load.
3. **Pub/Sub**: Enable real-time messaging or notifications for users.
4. **Job Queue**: Use Redis lists to queue background jobs like email sending or image processing.