Redis

> Important : The best way to learn Redis is by reading it's documentation , here is the link:

https://redis.io/docs/latest/develop/data-types/

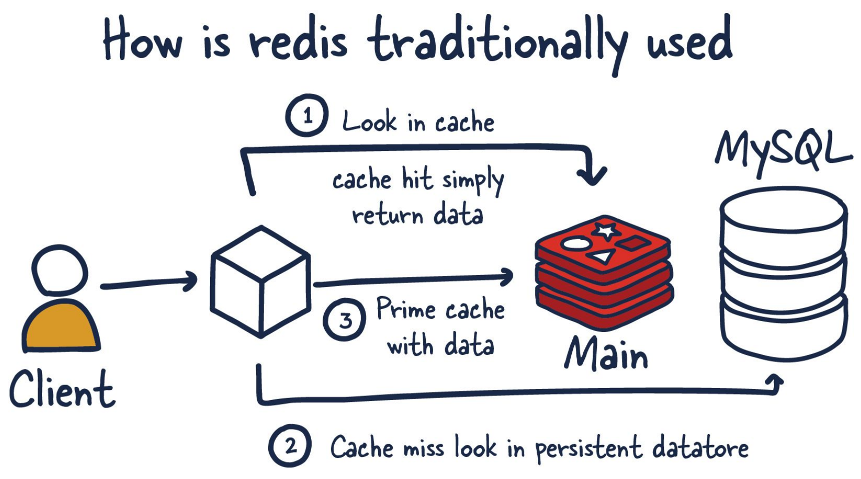
Section 1: Redis Basics

Starting with the Fundamentals

I started with understanding the basics of Redis, a powerful in-memory data structure store widely used as a database, cache, and message broker.

- **Introduction to Redis**: Learned about Redis's key-value store model and how it handles data in memory for ultra-fast access.
- **Setting Up Redis**: Gained hands-on experience setting up and configuring Redis on a local development environment using docker.

Redis, a NoSQL database, serves as an in-memory key-value store, known for its speed and efficiency. I began by understanding its architecture, which is crucial for leveraging its full potential in data-driven applications.


How is redis traditionally used
1 Look in cache
cache hit simply return data
MySQL
Client
3 Prime cache with data
Main
2 Cache miss look in persistent datatore

Installation

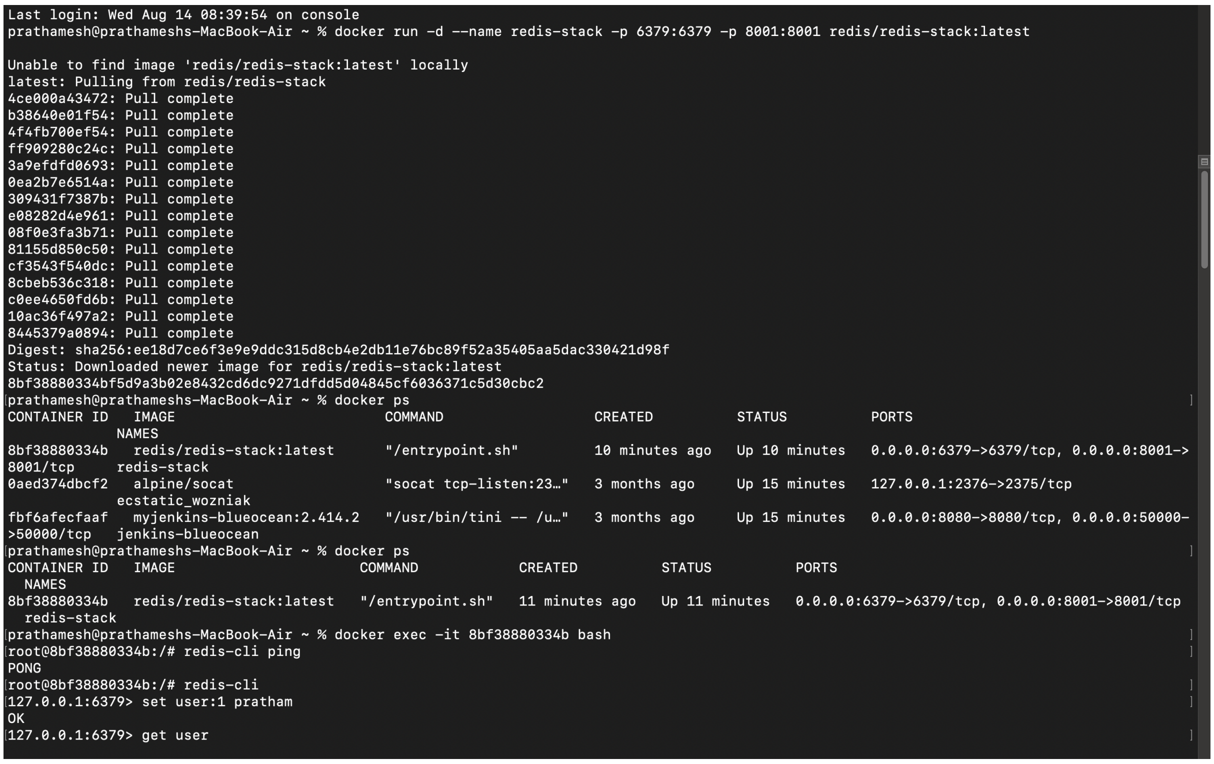
I have installed Redis using docker :

https://redis.io/docs/latest/operate/oss_and_stack/install/install-stack/docker/

To get started with Redis Stack using Docker, you first need to select a Docker image:

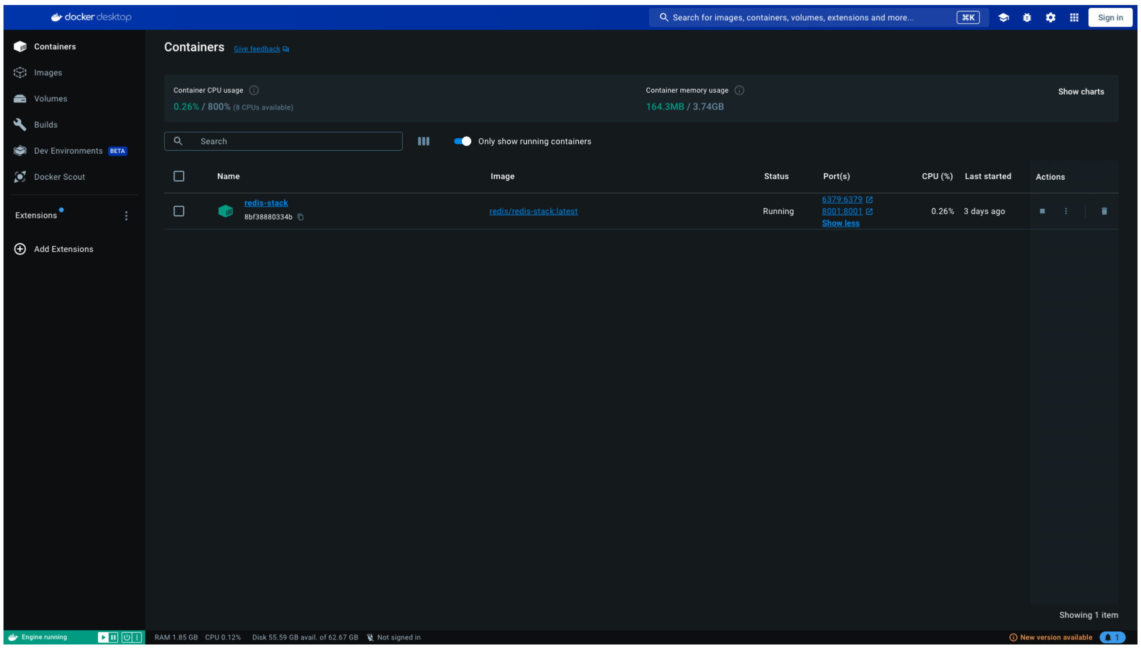
- `redis/redis-stack` contains both Redis Stack server and Redis Insight. This container is best for local development because you can use the embedded Redis Insight to visualize your data.

Using these commands we can install Redis using docker locally :



```
Last login: Wed Aug 14 08:39:54 on console
prathamesh@prathameshs-MacBook-Air ~ % docker run -d --name redis-stack -p 6379:6379 -p 8001:8001 redis/redis-stack:latest

Unable to find image 'redis/redis-stack:latest' locally
latest: Pulling from redis/redis-stack
4ce000a43472: Pull complete
b38640e01f54: Pull complete
4f4fb700ef54: Pull complete
ff909280c24c: Pull complete
3a9efdfd0693: Pull complete
0ea2b7e6514a: Pull complete
309431f7387b: Pull complete
e08282d4e961: Pull complete
08f0e3fa3b71: Pull complete
81155d850c50: Pull complete
cf3543f540dc: Pull complete
8cbeb536c318: Pull complete
c0ee4650fd6b: Pull complete
10ac36f497a2: Pull complete
8445379a0894: Pull complete
Digest: sha256:ee18d7ce6f3e9e9ddc315d8cb4e2db11e76bc89f52a35405aa5dac330421d98f
Status: Downloaded newer image for redis/redis-stack:latest
8bf38880334bf5d9a3b02e8432cd6dc9271dfdd5d04845cf6036371c5d30cbc2
prathamesh@prathameshs-MacBook-Air ~ % docker ps
CONTAINER ID   IMAGE                          COMMAND                  CREATED          STATUS          PORTS
             NAMES
8bf38880334b   redis/redis-stack:latest      "/entrypoint.sh"         10 minutes ago   Up 10 minutes   0.0.0.0:6379->6379/tcp, 0.0.0.0:8001->
8001/tcp     redis-stack
0aed374dbcf2   alpine/socat                   "socat tcp-listen:23…"   3 months ago     Up 15 minutes   127.0.0.1:2376->2375/tcp
             ecstatic_wozniak
fbf6afecfaaf   myjenkins-blueocean:2.414.2    "/usr/bin/tini -- /u…"   3 months ago     Up 15 minutes   0.0.0.0:8080->8080/tcp, 0.0.0.0:50000-
>50000/tcp   jenkins-blueocean
prathamesh@prathameshs-MacBook-Air ~ % docker ps
CONTAINER ID   IMAGE                      COMMAND            CREATED          STATUS          PORTS
  NAMES
8bf38880334b   redis/redis-stack:latest   "/entrypoint.sh"   11 minutes ago   Up 11 minutes   0.0.0.0:6379->6379/tcp, 0.0.0.0:8001->8001/tcp
  redis-stack
prathamesh@prathameshs-MacBook-Air ~ % docker exec -it 8bf38880334b bash
root@8bf38880334b:/# redis-cli ping
PONG
root@8bf38880334b:/# redis-cli
127.0.0.1:6379> set user:1 pratham
OK
127.0.0.1:6379> get user
```

Once you complete the setup it will be up and running on port 8001 :



Containers

Container CPU usage 0.26% / 800% (8 CPUs available)
Container memory usage 164.3MB / 3.74GB

Name	Image	Status	Port(s)	CPU (%)	Last started	Actions
redis-stack 8bf38880334b	redis/redis-stack:latest	Running	6379:6379 8001:8001 Show less	0.26%	3 days ago	

Showing 1 item

RAM 1.85 GB CPU 0.12% Disk 55.59 GB avail. of 62.67 GB Not signed in

Section 2: Exploring Redis Data Structures

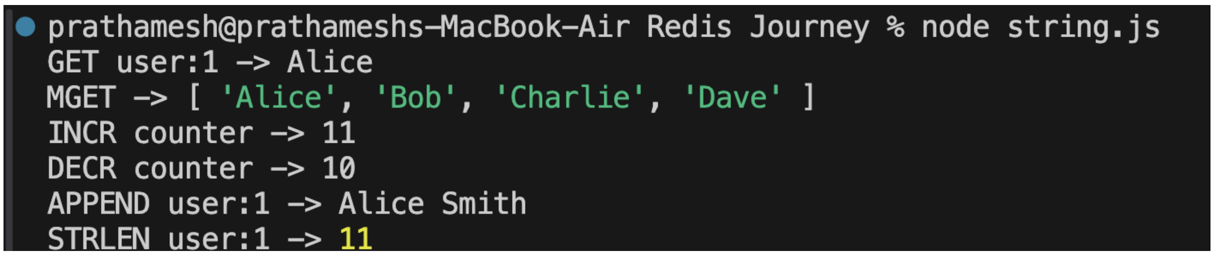
I have chosen node.js because it is very easy and faster , you can choose any .

Strings -

Here's a code snippet that covers basic string commands like `SET`, `GET`, `MSET`, `MGET`, `INCR`, `DECR`, `APPEND`, and `STRLEN`.

```
const client = require('./client');

client.on('error', (err) => {
  console.error('Redis error:', err);
});

async function init() {

  // SET a value
  await client.set('user:1', 'Alice');
  await client.set('user:2', 'Bob');

  // GET a value
  const user1 = await client.get('user:1');
  console.log("GET user:1 ->", user1); // Output: Alice

  // MSET multiple values
  await client.mset({
    'user:3': 'Charlie',
    'user:4': 'Dave'
  });

  // MGET multiple values
  const users = await client.mget('user:1', 'user:2', 'user:3', 'user:4');
  console.log("MGET ->", users); // Output: ['Alice', 'Bob', 'Charlie', 'Da
ve']

  // INCR (increment a value)
  await client.set('counter', 10);  // Set initial value
  await client.incr('counter');
  const counterAfterIncr = await client.get('counter');
  console.log("INCR counter ->", counterAfterIncr); // Output: 11

  // DECR (decrement a value)
  await client.decr('counter');
  const counterAfterDecr = await client.get('counter');
  console.log("DECR counter ->", counterAfterDecr); // Output: 10

  // APPEND (append a string to an existing value)
  await client.append('user:1', ' Smith');
  const user1AfterAppend = await client.get('user:1');
  console.log("APPEND user:1 ->", user1AfterAppend); // Output: Alice Smith

  // STRLEN (get the length of a string)
  const user1Length = await client.strlen('user:1');
  console.log("STRLEN user:1 ->", user1Length); // Output: 11
  // Clean up and disconnect
  await client.quit();
}

init().catch(console.error);
```

Output :



```
prathamesh@prathameshs-MacBook-Air Redis Journey % node string.js
GET user:1 -> Alice
MGET -> [ 'Alice', 'Bob', 'Charlie', 'Dave' ]
INCR counter -> 11
DECR counter -> 10
APPEND user:1 -> Alice Smith
STRLEN user:1 -> 11
```

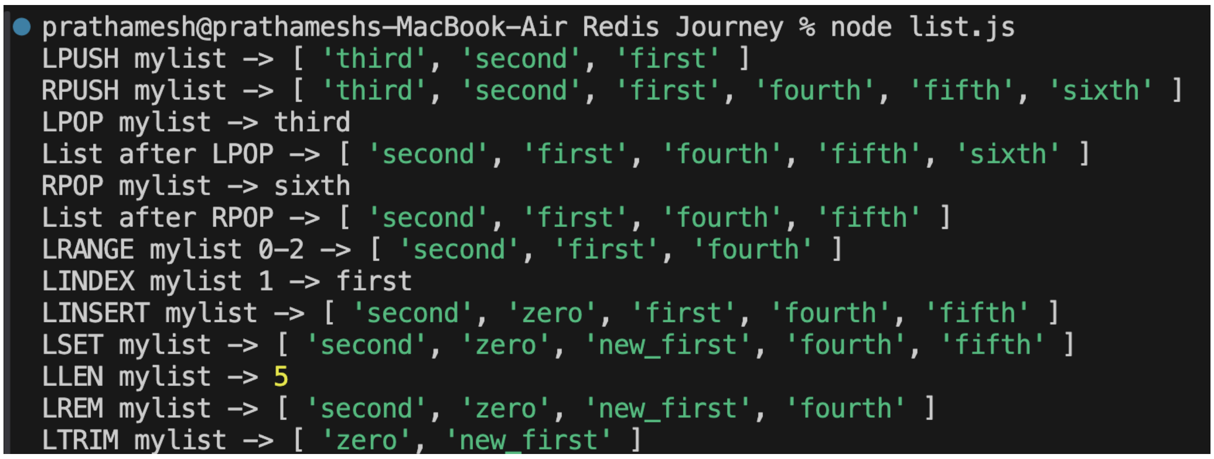
Lists -

Redis lists are ordered collections of strings, and they support operations like pushing elements to the head or tail, popping elements, getting elements by index, trimming the list, and more.

```
const Redis = require('ioredis');
const client = new Redis();

client.on('error', (err) => {
  console.error('Redis error:', err);
});

async function init() {

  // LPUSH (push one or more elements to the head of the list)
  await client.lpush('mylist', 'first');
  await client.lpush('mylist', 'second', 'third');
  const listAfterLPush = await client.lrange('mylist', 0, -1);
  console.log("LPUSH mylist ->", listAfterLPush); // Output: ['third', 'sec
ond', 'first']

  // RPUSH (push one or more elements to the tail of the list)
  await client.rpush('mylist', 'fourth');
  await client.rpush('mylist', 'fifth', 'sixth');
  const listAfterRPush = await client.lrange('mylist', 0, -1);
  console.log("RPUSH mylist ->", listAfterRPush); // Output: ['third', 'sec
ond', 'first', 'fourth', 'fifth', 'sixth']

  // LPOP (pop an element from the head of the list)
  const lpopValue = await client.lpop('mylist');
  console.log("LPOP mylist ->", lpopValue); // Output: 'third'
  const listAfterLPop = await client.lrange('mylist', 0, -1);
  console.log("List after LPOP ->", listAfterLPop); // Output: ['second',
'first', 'fourth', 'fifth', 'sixth']

  // RPOP (pop an element from the tail of the list)
  const rpopValue = await client.rpop('mylist');
  console.log("RPOP mylist ->", rpopValue); // Output: 'sixth'
  const listAfterRPop = await client.lrange('mylist', 0, -1);
  console.log("List after RPOP ->", listAfterRPop); // Output: ['second',
'first', 'fourth', 'fifth']

  // LRANGE (get a range of elements from the list)
  const listRange = await client.lrange('mylist', 0, 2);
  console.log("LRANGE mylist 0-2 ->", listRange); // Output: ['second', 'fi
rst', 'fourth']

  // LINDEX (get an element by its index)
  const lindexValue = await client.lindex('mylist', 1);
  console.log("LINDEX mylist 1 ->", lindexValue); // Output: 'first'

  // LINSERT (insert an element before or after another element)
  await client.linsert('mylist', 'BEFORE', 'first', 'zero');
  const listAfterLInsert = await client.lrange('mylist', 0, -1);
  console.log("LINSERT mylist ->", listAfterLInsert); // Output: ['second',
'zero', 'first', 'fourth', 'fifth']

  // LSET (set the value of an element at a specific index)
  await client.lset('mylist', 2, 'new_first');
  const listAfterLSet = await client.lrange('mylist', 0, -1);
  console.log("LSET mylist ->", listAfterLSet); // Output: ['second', 'zer
o', 'new_first', 'fourth', 'fifth']

  // LLEN (get the length of the list)
  const listLength = await client.llen('mylist');
  console.log("LLEN mylist ->", listLength); // Output: 5

  // LREM (remove elements from the list)
  await client.lrem('mylist', 1, 'fifth');
  const listAfterLRem = await client.lrange('mylist', 0, -1);
  console.log("LREM mylist ->", listAfterLRem); // Output: ['second', 'zer
o', 'new_first', 'fourth']

  // LTRIM (trim the list to the specified range)
  await client.ltrim('mylist', 1, 2);
  const listAfterLTrim = await client.lrange('mylist', 0, -1);
  console.log("LTRIM mylist ->", listAfterLTrim); // Output: ['zero', 'new_
first']

  // Clean up and disconnect
  await client.quit();
```

```
}

init().catch(console.error);
```

Output :



```
prathamesh@prathameshs-MacBook-Air Redis Journey % node list.js
LPUSH mylist -> [ 'third', 'second', 'first' ]
RPUSH mylist -> [ 'third', 'second', 'first', 'fourth', 'fifth', 'sixth' ]
LPOP mylist -> third
List after LPOP -> [ 'second', 'first', 'fourth', 'fifth', 'sixth' ]
RPOP mylist -> sixth
List after RPOP -> [ 'second', 'first', 'fourth', 'fifth' ]
LRANGE mylist 0-2 -> [ 'second', 'first', 'fourth' ]
LINDEX mylist 1 -> first
LINSERT mylist -> [ 'second', 'zero', 'first', 'fourth', 'fifth' ]
LSET mylist -> [ 'second', 'zero', 'new_first', 'fourth', 'fifth' ]
LLEN mylist -> 5
LREM mylist -> [ 'second', 'zero', 'new_first', 'fourth' ]
LTRIM mylist -> [ 'zero', 'new_first' ]
```

Blocking commands -

Redis provides blocking list operations such as `BLPOP` and `BRPOP`, which block the connection until an element is available to pop from the list. These functions are useful when you want to implement a producer-consumer pattern or when you need to wait for elements to be added to the list.

Here's a demonstration of blocking functions in Redis lists:

```
const redis = require('redis');

// Create a Redis client
const client = redis.createClient();

client.on('error', (err) => {
  console.error('Redis error:', err);
});

async function blockingListOps() {
  // Connect to the Redis server
  await client.connect();

  // Producer: Adds elements to the list using RPUSH
  async function producer() {
    await new Promise(resolve => setTimeout(resolve, 2000)); // Simulate so
me delay
    console.log("Producer: Adding elements to the list...");
    await client.rPush('mylist', 'element1');
    await client.rPush('mylist', 'element2');
    await client.rPush('mylist', 'element3');
    console.log("Producer: Elements added to the list");
  }
  // Consumer: Waits for elements to be available using BLPOP
  async function consumer() {
    console.log("Consumer: Waiting for an element...");
    const result = await client.blPop('mylist', 0); // Block indefinitely until an element is available
    console.log("Consumer: Popped element ->", result); // Output: The first element from the list
  }

  // Start the consumer and producer
  consumer();  // Consumer will start waiting for elements immediately
  await producer();  // Producer will add elements after a delay

  // Close the connection after some delay to allow the operations to complete
  setTimeout(async () => {
    await client.quit();
  }, 5000);
}

blockingListOps().catch(console.error);
```

Key Points:

- `BLPOP`: This function removes and returns the first element of the list, blocking until an element is available. If you pass a timeout (in seconds), it will wait until the timeout expires and return `null` if no element is available. Passing `0` as the timeout makes it block indefinitely.
- `BRPOP`: Similar to `BLPOP`, but it removes and returns the last element of the list.

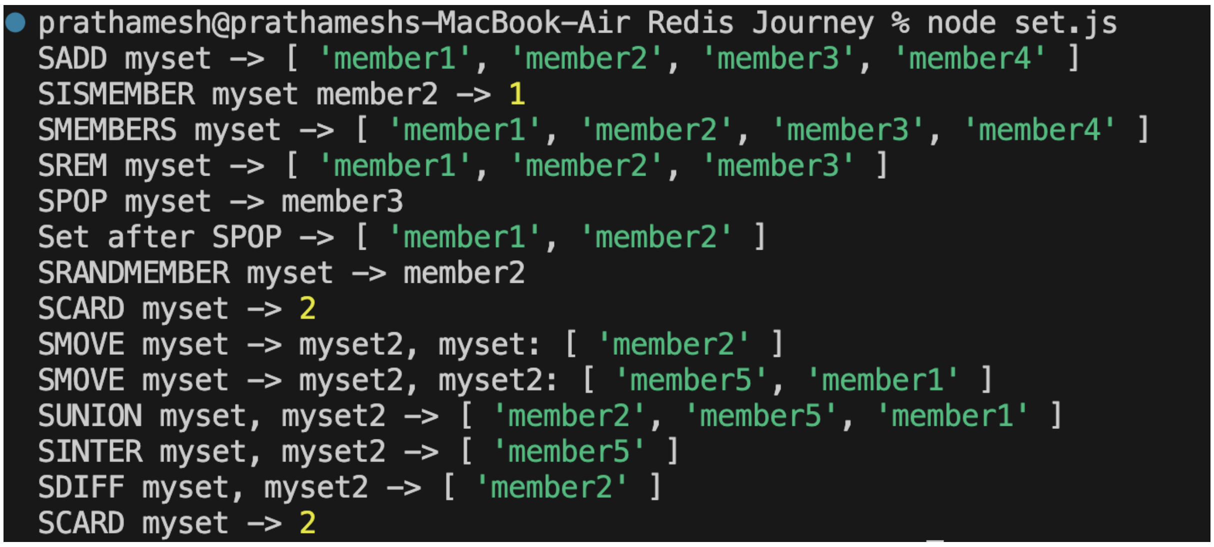
Sets -

Redis sets are unordered collections of unique strings. They support various operations like adding, removing, and testing for the existence of members, as well as performing set operations like union, intersection, and difference.

```
const client = require('./client'); // This imports the 'ioredis' client instance

client.on('error', (err) => {
  console.error('Redis error:', err);
});

async function init() {
  // SADD (Add one or more members to a set)
  await client.sadd('myset', 'member1');
  await client.sadd('myset', 'member2', 'member3', 'member4');
  console.log("SADD myset ->", await client.smembers('myset')); // Output: ['member1', 'member2', 'member3', 'member4']
  // SISMEMBER (Check if a member exists in the set)
  const isMember = await client.sismember('myset', 'member2');
  console.log("SISMEMBER myset member2 ->", isMember); // Output: 1 (true)

  // SMEMBERS (Get all the members of a set)
  const members = await client.smembers('myset');
  console.log("SMEMBERS myset ->", members); // Output: ['member1', 'member
2', 'member3', 'member4']

  // SREM (Remove one or more members from a set)
  await client.srem('myset', 'member4');
  console.log("SREM myset ->", await client.smembers('myset')); // Output:
['member1', 'member2', 'member3']

  // SPOP (Remove and return a random member from the set)
  const poppedMember = await client.spop('myset');
  console.log("SPOP myset ->", poppedMember); // Output: A random member
(e.g., 'member3')
  console.log("Set after SPOP ->", await client.smembers('myset'));

  // SRANDMEMBER (Get a random member from the set without removing it)
  const randomMember = await client.srandmember('myset');
  console.log("SRANDMEMBER myset ->", randomMember); // Output: A random me
mber (e.g., 'member1')

  // SCARD (Get the number of members in the set)
  const setSize = await client.scard('myset');
  console.log("SCARD myset ->", setSize); // Output: The size of the set
(e.g., 2)

  // SMOVE (Move a member from one set to another)
  await client.sadd('myset2', 'member5');
  await client.smove('myset', 'myset2', 'member1');
  console.log("SMOVE myset -> myset2, myset:", await client.smembers('myse
t')); // Output: Remaining members in 'myset'
  console.log("SMOVE myset -> myset2, myset2:", await client.smembers('myse
t2')); // Output: Members in 'myset2'

  // SUNION (Get the union of multiple sets)
  const unionSet = await client.sunion('myset', 'myset2');
  console.log("SUNION myset, myset2 ->", unionSet); // Output: Union of 'my
set' and 'myset2'

  // SINTER (Get the intersection of multiple sets)
  await client.sadd('myset', 'member5');
  const intersectSet = await client.sinter('myset', 'myset2');
  console.log("SINTER myset, myset2 ->", intersectSet); // Output: Intersec
```

```
tion of 'myset' and 'myset2'

  // SDIFF (Get the difference of multiple sets)
  const diffSet = await client.sdiff('myset', 'myset2');
  console.log("SDIFF myset, myset2 ->", diffSet); // Output: Difference of
'myset' and 'myset2'

  // SCARD (Get the number of members in a set)
  const card = await client.scard('myset');
  console.log("SCARD myset ->", card); // Output: Number of members in 'mys
et'

  // Clean up and disconnect
  await client.del('myset', 'myset2');
  await client.quit();
}

init().catch(console.error);
```

Output :



```
prathamesh@prathameshs-MacBook-Air Redis Journey % node set.js
SADD myset -> [ 'member1', 'member2', 'member3', 'member4' ]
SISMEMBER myset member2 -> 1
SMEMBERS myset -> [ 'member1', 'member2', 'member3', 'member4' ]
SREM myset -> [ 'member1', 'member2', 'member3' ]
SPOP myset -> member3
Set after SPOP -> [ 'member1', 'member2' ]
SRANDMEMBER myset -> member2
SCARD myset -> 2
SMOVE myset -> myset2, myset: [ 'member2' ]
SMOVE myset -> myset2, myset2: [ 'member5', 'member1' ]
SUNION myset, myset2 -> [ 'member2', 'member5', 'member1' ]
SINTER myset, myset2 -> [ 'member5' ]
SDIFF myset, myset2 -> [ 'member2' ]
SCARD myset -> 2
```

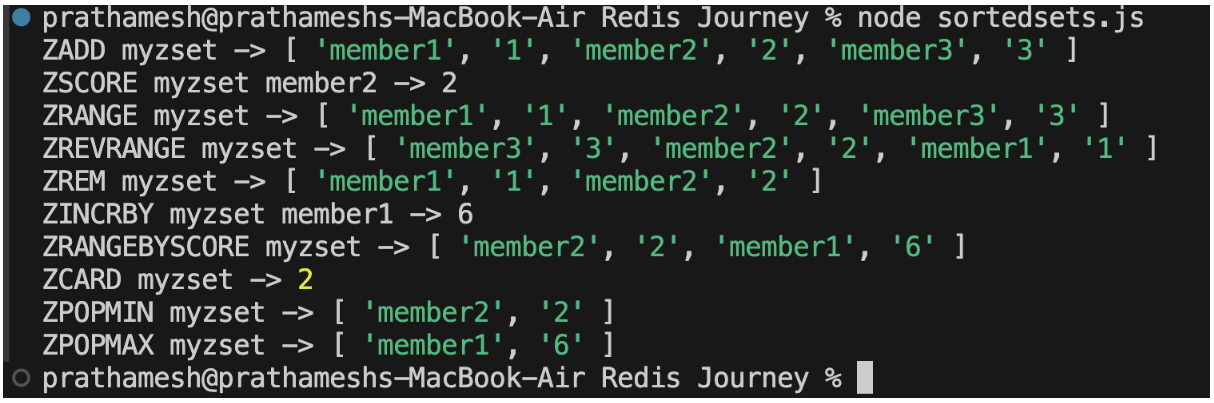
Sorted Sets(Priority Queues) -

Redis sorted sets are similar to regular sets but with an additional score associated with each member, which allows them to be sorted. This makes them ideal for implementing priority queues and other ordered collections.

```
const client = require('./client'); // This imports the 'ioredis' client in
stance

client.on('error', (err) => {
  console.error('Redis error:', err);
});
```

```
async function init() {
  // ZADD (Add one or more members to a sorted set, or update the score if
it already exists)
  await client.zadd('myzset', 1, 'member1');
  await client.zadd('myzset', 2, 'member2');
  await client.zadd('myzset', 3, 'member3');
  console.log("ZADD myzset ->", await client.zrange('myzset', 0, -1, 'withs
cores'));
  // Output: [ 'member1', '1', 'member2', '2', 'member3', '3' ]

  // ZSCORE (Get the score of a member in a sorted set)
  const score = await client.zscore('myzset', 'member2');
  console.log("ZSCORE myzset member2 ->", score); // Output: 2

  // ZRANGE (Get all the members in a sorted set within a given score rang
e, with optional withscores)
  const members = await client.zrange('myzset', 0, -1, 'withscores');
  console.log("ZRANGE myzset ->", members);
  // Output: [ 'member1', '1', 'member2', '2', 'member3', '3' ]

  // ZREVRANGE (Get all the members in a sorted set within a given range, s
orted from highest to lowest score)
  const reversedMembers = await client.zrevrange('myzset', 0, -1, 'withscor
es');
  console.log("ZREVRANGE myzset ->", reversedMembers);
  // Output: [ 'member3', '3', 'member2', '2', 'member1', '1' ]

  // ZREM (Remove one or more members from a sorted set)
  await client.zrem('myzset', 'member3');
  console.log("ZREM myzset ->", await client.zrange('myzset', 0, -1, 'withs
cores'));
  // Output: [ 'member1', '1', 'member2', '2' ]

  // ZINCRBY (Increment the score of a member in a sorted set)
  await client.zincrby('myzset', 5, 'member1');
  console.log("ZINCRBY myzset member1 ->", await client.zscore('myzset', 'm
ember1'));
  // Output: 6

  // ZRANGEBYSCORE (Get all the members in a sorted set within a given scor
e range)
  const rangeByScore = await client.zrangebyscore('myzset', 1, 10, 'withsco
res');
  console.log("ZRANGEBYSCORE myzset ->", rangeByScore);
  // Output: [ 'member1', '6', 'member2', '2' ]

  // ZCARD (Get the number of members in a sorted set)
  const card = await client.zcard('myzset');
  console.log("ZCARD myzset ->", card); // Output: Number of members in the
sorted set

  // ZPOPMIN (Remove and return the member with the lowest score from a sor
ted set)
  const poppedMin = await client.zpopmin('myzset');
  console.log("ZPOPMIN myzset ->", poppedMin);
  // Output: [ [ 'member2', '2' ] ]

  // ZPOPMAX (Remove and return the member with the highest score from a so
rted set)
  const poppedMax = await client.zpopmax('myzset');
  console.log("ZPOPMAX myzset ->", poppedMax);
  // Output: [ [ 'member1', '6' ] ]

  // Clean up and disconnect
  await client.del('myzset');
  await client.quit();
}

init().catch(console.error);
```

Output :



```
prathamesh@prathameshs-MacBook-Air Redis Journey % node sortedsets.js
ZADD myzset -> [ 'member1', '1', 'member2', '2', 'member3', '3' ]
ZSCORE myzset member2 -> 2
ZRANGE myzset -> [ 'member1', '1', 'member2', '2', 'member3', '3' ]
ZREVRANGE myzset -> [ 'member3', '3', 'member2', '2', 'member1', '1' ]
ZREM myzset -> [ 'member1', '1', 'member2', '2' ]
ZINCRBY myzset member1 -> 6
ZRANGEBYSCORE myzset -> [ 'member2', '2', 'member1', '6' ]
ZCARD myzset -> 2
ZPOPMIN myzset -> [ 'member2', '2' ]
ZPOPMAX myzset -> [ 'member1', '6' ]
prathamesh@prathameshs-MacBook-Air Redis Journey %
```

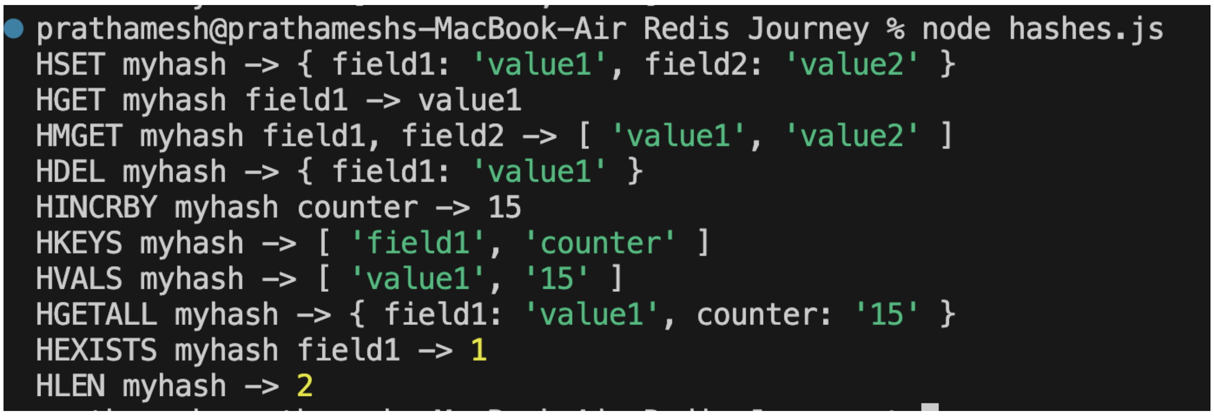
Hashes -

Redis hashes are maps between string field and string values, and are ideal for storing objects.

```
const client = require('./client'); // This imports the 'ioredis' client in
stance

client.on('error', (err) => {
  console.error('Redis error:', err);
});

async function init() {
  // HSET (Set the string value of a hash field)
  await client.hset('myhash', 'field1', 'value1');
  await client.hset('myhash', 'field2', 'value2');
  console.log("HSET myhash ->", await client.hgetall('myhash')); // Output:
{ field1: 'value1', field2: 'value2' }

  // HGET (Get the value of a hash field)
  const field1Value = await client.hget('myhash', 'field1');
  console.log("HGET myhash field1 ->", field1Value); // Output: 'value1'

  // HMGET (Get the values of multiple hash fields)
  const values = await client.hmget('myhash', 'field1', 'field2');
  console.log("HMGET myhash field1, field2 ->", values); // Output: ['value
1', 'value2']

  // HDEL (Delete one or more hash fields)
  await client.hdel('myhash', 'field2');
  console.log("HDEL myhash ->", await client.hgetall('myhash')); // Output:
{ field1: 'value1' }

  // HINCRBY (Increment the integer value of a hash field by the given numb
er)
  await client.hset('myhash', 'counter', 10);
  await client.hincrby('myhash', 'counter', 5);
  console.log("HINCRBY myhash counter ->", await client.hget('myhash', 'cou
nter')); // Output: 15

  // HKEYS (Get all the fields in a hash)
  const fields = await client.hkeys('myhash');
  console.log("HKEYS myhash ->", fields); // Output: ['field1', 'counter']

  // HVALS (Get all the values in a hash)
  const valuesInHash = await client.hvals('myhash');
  console.log("HVALS myhash ->", valuesInHash); // Output: ['value1', '15']

  // HGETALL (Get all the fields and values in a hash)
  const hashContents = await client.hgetall('myhash');
  console.log("HGETALL myhash ->", hashContents); // Output: { field1: 'val
ue1', counter: '15' }

  // HEXISTS (Check if a hash field exists)
  const fieldExists = await client.hexists('myhash', 'field1');
  console.log("HEXISTS myhash field1 ->", fieldExists); // Output: 1 (true)

  // HLEN (Get the number of fields in a hash)
  const hashLength = await client.hlen('myhash');
  console.log("HLEN myhash ->", hashLength); // Output: Number of fields in
the hash
```

```
  // Clean up and disconnect
  await client.del('myhash');
  await client.quit();
}

init().catch(console.error);
```

Output :



```
prathamesh@prathameshs-MacBook-Air Redis Journey % node hashes.js
HSET myhash -> { field1: 'value1', field2: 'value2' }
HGET myhash field1 -> value1
HMGET myhash field1, field2 -> [ 'value1', 'value2' ]
HDEL myhash -> { field1: 'value1' }
HINCRBY myhash counter -> 15
HKEYS myhash -> [ 'field1', 'counter' ]
HVALS myhash -> [ 'value1', '15' ]
HGETALL myhash -> { field1: 'value1', counter: '15' }
HEXISTS myhash field1 -> 1
HLEN myhash -> 2
```

Section 3: Advanced Redis Features

Streams -

Redis Streams are a powerful feature that allows you to handle real-time data streams. They provide a way to manage data that continuously arrives, such as log entries, messages, or events. Redis Streams support features like data persistence, message acknowledgement, and consumer groups.

```
const client = require('./client'); // This imports the 'ioredis' client in
stance

client.on('error', (err) => {
  console.error('Redis error:', err);
});

async function init() {
  // XADD (Add a new entry to the stream)
  await client.xadd('mystream', '*', 'field1', 'value1');
  await client.xadd('mystream', '*', 'field2', 'value2');
  console.log("XADD mystream ->", await client.xrange('mystream', 0, -1));

  // XRANGE (Retrieve entries from the stream within a given range)
  const range = await client.xrange('mystream', 0, -1);
  console.log("XRANGE mystream ->", range);

  // XREVRANGE (Retrieve entries from the stream within a given range in re
verse order)
```

```
  const revRange = await client.xrevrange('mystream', '+', '-', 'COUNT', 1
0);
  console.log("XREVRANGE mystream ->", revRange);

  // XREAD (Read data from one or multiple streams, blocking or non-blockin
g)
  const xread = await client.xread('BLOCK', 0, 'STREAMS', 'mystream', '$');
  console.log("XREAD mystream ->", xread);

  // XDEL (Delete specific entries from a stream by ID)
  const [stream, count] = await client.xdel('mystream', '1681284007233-0');
// Use actual entry IDs here
  console.log(`XDEL mystream -> Deleted ${count} entries`);

  // XACK (Acknowledge one or more messages as processed by a consumer grou
p)
  // Create a consumer group
  await client.xgroup('CREATE', 'mystream', 'mygroup', '$', 'MKSTREAM');
  await client.xack('mystream', 'mygroup', '1681284007233-0');
  console.log("XACK mystream -> Acknowledged message");

  // XGROUP (Create, destroy, or set the configuration of a consumer group)
  await client.xgroup('DESTROY', 'mystream', 'mygroup');
  console.log("XGROUP mystream -> Destroyed the consumer group");

  // XTRIM (Trim the stream to the specified length)
  await client.xtrim('mystream', 5, 'MAXLEN');
  console.log("XTRIM mystream -> Trimmed the stream");

  // Clean up and disconnect
  await client.del('mystream');
  await client.quit();
}

init().catch(console.error);
```

Pub/Sub (Publish/Subscribe) :

- *Use Cases*: Real-time messaging, notifications, and event-driven systems.
- *Key Functions*: `PUBLISH`, `SUBSCRIBE`, `UNSUBSCRIBE`.
- *Learning Outcome*: Implemented a basic messaging system using Redis Pub/Sub, enabling real-time notifications across services.

Basic Workflow

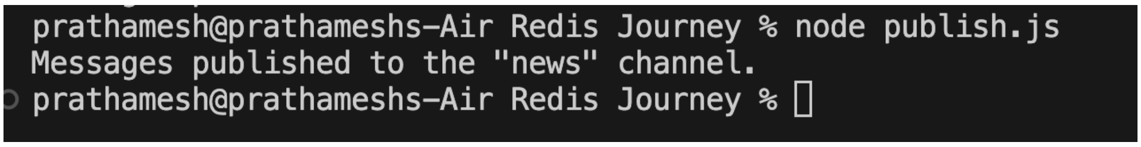
1. **Publisher:** A publisher sends messages to a specific channel. Any message published to a channel is immediately available to all subscribers of that channel.

2. **Subscriber:** A subscriber listens to a specific channel (or channels). When a message is published to that channel, the subscriber receives and processes the message.
3. **Channels:** Channels are the medium through which messages are passed. Both the publisher and subscriber must reference the same channel for the communication to work.

Publish.js -

```
const client = require('./client'); // This imports the 'ioredis' client in
stance

client.on('error', (err) => {
  console.error('Redis error:', err);
});

async function publish() {
  // Publish messages to a channel
  await client.publish('news', 'Hello, Redis Pub/Sub!');
  await client.publish('news', 'Another message on the news channel.');

  console.log('Messages published to the "news" channel.');

  // Clean up and disconnect
  await client.quit();
}

publish().catch(console.error);
```

Output :

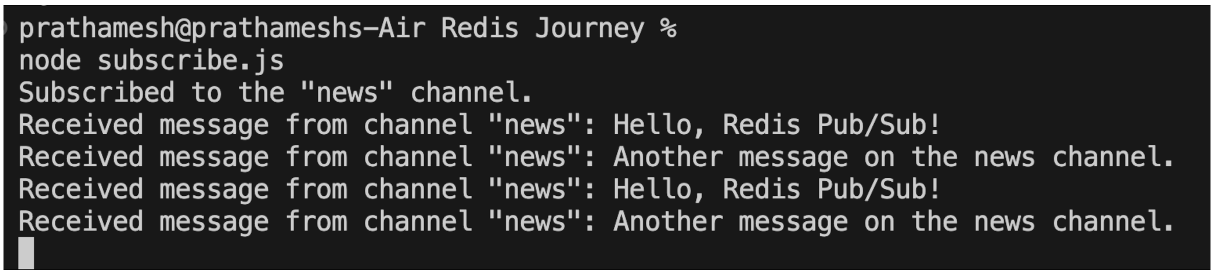


```
prathamesh@prathameshs-Air Redis Journey % node publish.js
Messages published to the "news" channel.
prathamesh@prathameshs-Air Redis Journey %
```

Subscribe.js -

```
const client = require('./client'); // This imports the 'ioredis' client in
stance
const {Redis} = require('ioredis');

client.on('error', (err) => {
  console.error('Redis error:', err);
});

async function subscribe() {
  // Create a subscriber client
  const subscriber = new Redis(); // Create a new instance for subscribing
  subscriber.on('message', (channel, message) => {
    console.log(`Received message from channel "${channel}": ${message}`);
  });

  // Subscribe to the channel
  await subscriber.subscribe('news');
  console.log('Subscribed to the "news" channel.');

  // Keep the process alive to receive messages
  process.on('SIGINT', async () => {
    console.log('Shutting down...');
    await subscriber.unsubscribe('news');
    await subscriber.quit();
    process.exit();
  });
}

subscribe().catch(console.error);
```

Output :



```
prathamesh@prathameshs-Air Redis Journey %
node subscribe.js
Subscribed to the "news" channel.
Received message from channel "news": Hello, Redis Pub/Sub!
Received message from channel "news": Another message on the news channel.
Received message from channel "news": Hello, Redis Pub/Sub!
Received message from channel "news": Another message on the news channel.
```