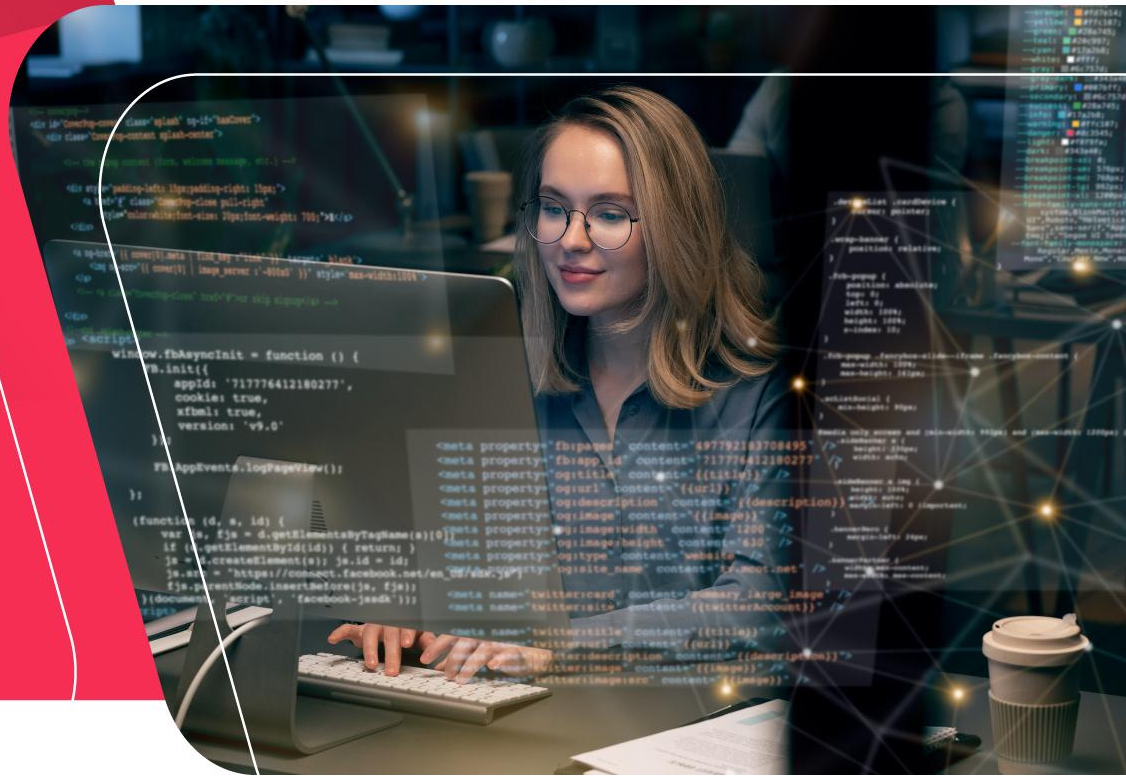
TIA-Academy

Java Bootcamp

Day 38



Technologies will be Use

- JDK 8/**11**/15
- JRE 8/**11**/15
- **Intellij IDEA Community Edition**
- JAVA 3rd Party Library (Network, DB, etc)
- JAVA Framework (Spring Boot & Spring JDBC)
- **MySQL Server Community**

TIA-Academy

REDIS

Redis Tutorial

- Redis tutorial provides basic and advanced concepts of Redis Database. Our Redis tutorial is designed for beginners and professionals both.
- Redis is a No SQL database which works on the concept of key-value pair.
- Our Redis Tutorial includes all topics of Redis such as what is redis, how to install redis, redis commands, data types, keys, lists, strings, hashes, sets, sorted sets, transaction, scripting, connection, server, interview questions etc

What is Redis

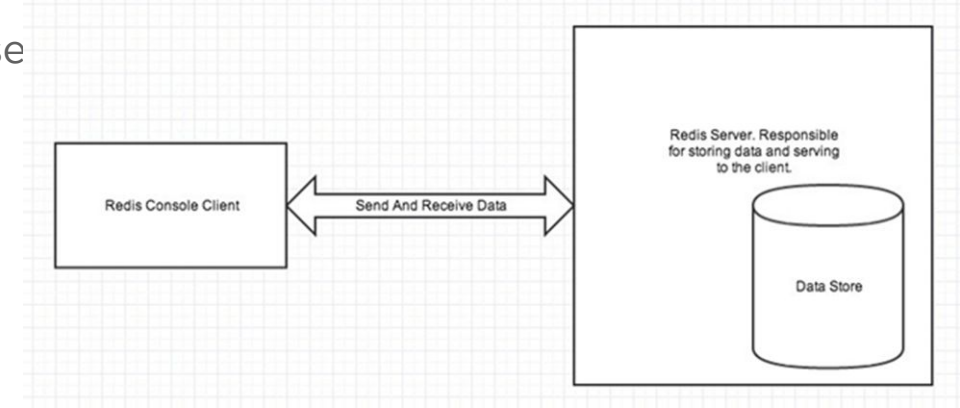
- Redis is a NoSQL database which follows the principle of key-value store. The key-value store provides ability to store some data called a value, inside a key. You can recieve this data later only if you know the exact key used to store it.

- Redis is a flexible, open-source (BSD licensed), in-memory data structure store, used as database, cache, and message broker. Redis is a NoSQL database so it facilitates users to store huge amount of data without the limit of a Relational database.

- Redis supports various types of data structures like strings, hashes, lists, sets, sorted sets, bitmaps, hyperloglogs and geospatial indexes with radius queries.

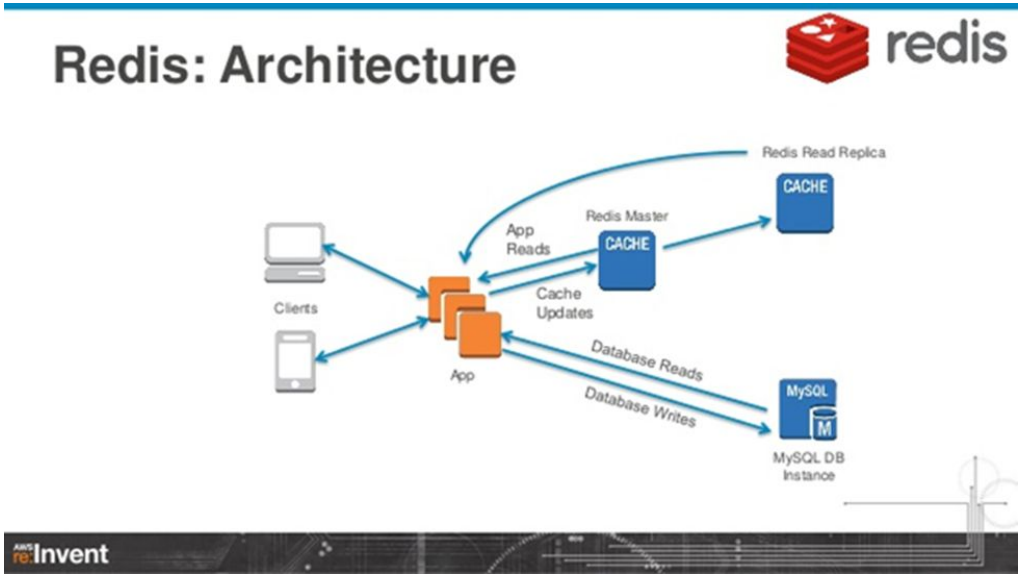
Redis Architecture

- There are two main processes in Redis architecture:
  - Redis Client
  - Redis Server
- These client and se nputers.



Redis Architecture

- Redis server is used to store data in memory . It controls all type of management and forms the main part of the architecture. You can create a Redis client or Redis console client when you install Redis application or you can use



Features of Redis

- **Speed:** Redis stores the whole dataset in primary memory that's why it is extremely fast. It loads up to 110,000 SETs/second and 81,000 GETs/second can be retrieved in an entry level Linux box. Redis supports Pipelining of commands and facilitates you to use multiple values in a single command to speed up communication with the client libraries.
- **Persistence:** While all the data lives in memory, changes are asynchronously saved on disk using flexible policies based on elapsed time and/or number of updates since last save. Redis supports an append-only file persistence mode. Check more on Persistence, or read the AppendOnlyFileHowto for more information.
- **Data Structures:** Redis supports various types of data structures such as strings, hashes, sets, lists, sorted sets with range queries, bitmaps, hyperloglogs and geospatial indexes with radius queries.
- **Atomic Operations:** Redis operations working on the different Data Types are atomic, so it is safe to set or increase a key, add and remove elements from a set, increase a counter etc.

Features of Redis

- **Supported Languages:** Redis supports a lot of languages such as ActionScript, C, C++, C#, Clojure, Common Lisp, D, Dart, Erlang, Go, Haskell, Haxe, Io, Java, JavaScript (Node.js), Julia, Lua, Objective-C, Perl, PHP, Pure Data, Python, R, Racket, Ruby, Rust, Scala, Smalltalk and Tcl.
- **Master/Slave Replication:** Redis follows a very simple and fast Master/Slave replication. It takes only one line in the configuration file to set it up, and 21 seconds for a Slave to complete the initial sync of 10 MM key set on an Amazon EC2 instance.
- **Sharding:** Redis supports sharding. It is very easy to distribute the dataset across multiple Redis instances, like other key-value store.
- **Portable:** Redis is written in ANSI C and works in most POSIX systems like Linux, BSD, Mac OS X, Solaris, and so on. Redis is reported to compile and work under WIN32 if compiled with Cygwin, but there is no official support for Windows currently.

Redis vs RDBMS

- Following is a list of differences between Redis and RDBMS:

Redis	RDBMS
Redis stores everything in primary memory.	RDBMS stores everything in secondary memory.
In Redis, Read and Write operations are extremely fast because of storing data in primary memory.	In RDBMS, Read and Write operations are slow because of storing data in secondary memory.
Primary memory is in lesser in size and much expensive than secondary so, Redis cannot store large files or binary data.	Secondary memory is in abundant in size and cheap than primary memory so, RDBMS can easily deal with these type of files.
Redis is used only to store those small textual information which needs to be accessed, modified and inserted at a very fast rate. If you try to write bulk data more than the available memory then you will receive errors.	RDBMS can hold large data which has less frequently usage and not required to be very fast.

Redis vs Other Key-Value Stores

Key-value store is a special type of database storage system where data is stored in form of key and value pairs.

Redis is different compared to other key-value stores because of the following:

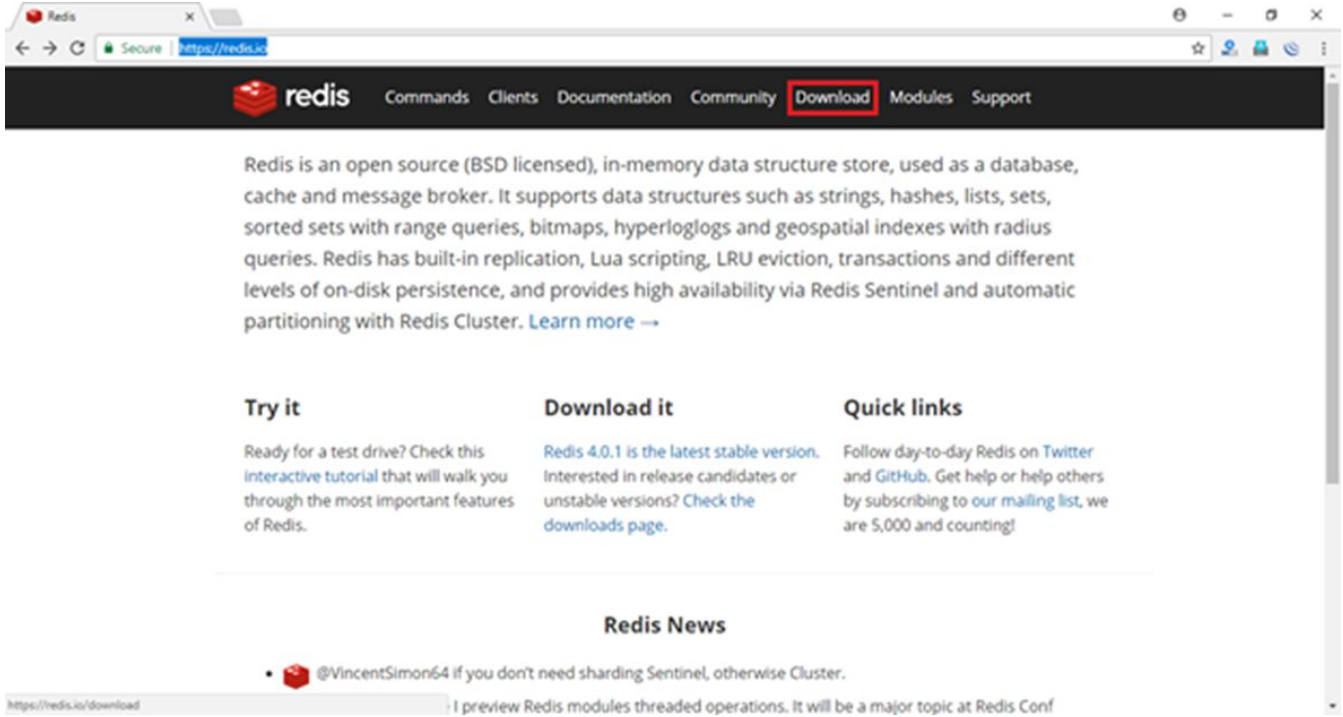
- Redis is a different evolution path in the key-value databases where values can contain more complex data types, with atomic operations defined on those data types.
- Redis data types are closely related to fundamental data structures and are exposed to the programmer as such, without additional abstraction layers.

Redis vs Other Key-Value Stores

- Redis is an in-memory but persistent on disk database, so it represents a different trade off where very high write and read speed is achieved with the limitation of data sets that can't be larger than memory.
- Another advantage of in memory databases is that the memory representation of complex data structures is much simpler to manipulate compared to the same data structure on disk, so Redis can do a lot, with little internal complexity.
- At the same time the two on-disk storage formats (RDB and AOF) don't need to be suitable for random access, so they are compact and always generated in an append-only fashion.

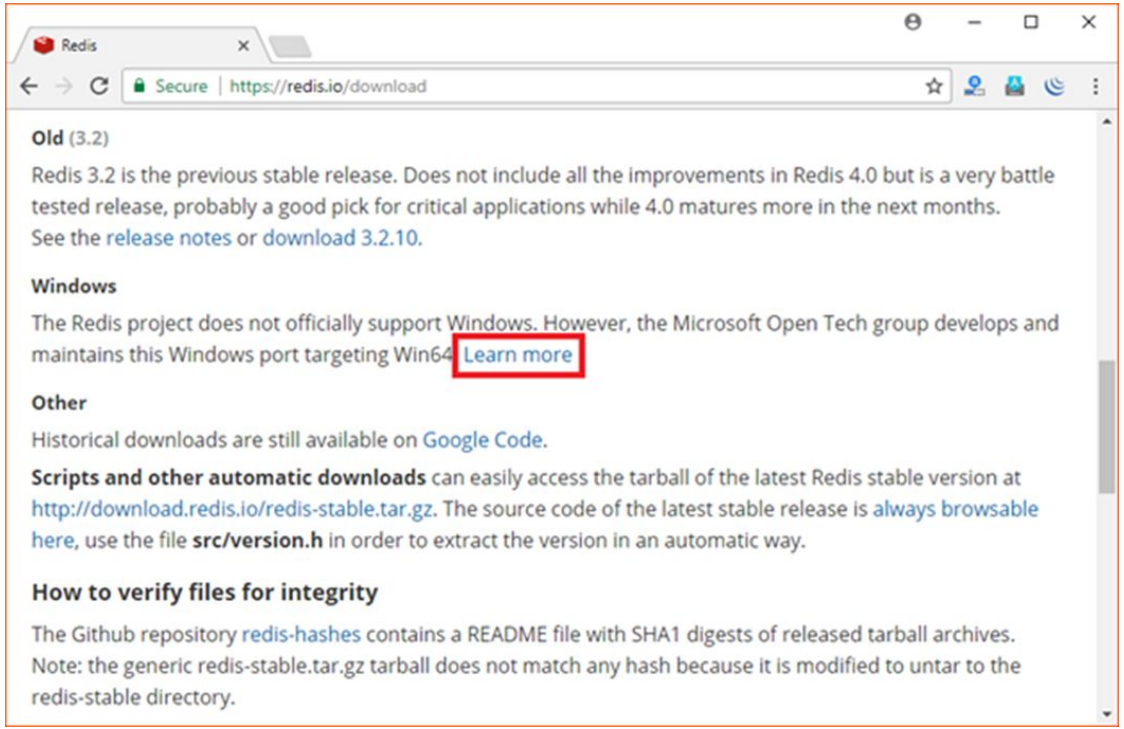
Redis Installation

- Go to Redis official website https://redis.io/ and follow images given below:



Redis Installation

- Scroll down to Windows section. You will see learn more. Click on it.



Redis Installation

- You will get release page:

GitHub - MicrosoftArchi

GitHub, Inc. [US] | https://github.com/MicrosoftArchive/redis

build passing | nuget v3.0.503 | chocolatey v3.0.503 | chocolatey 57k

Redis on Windows

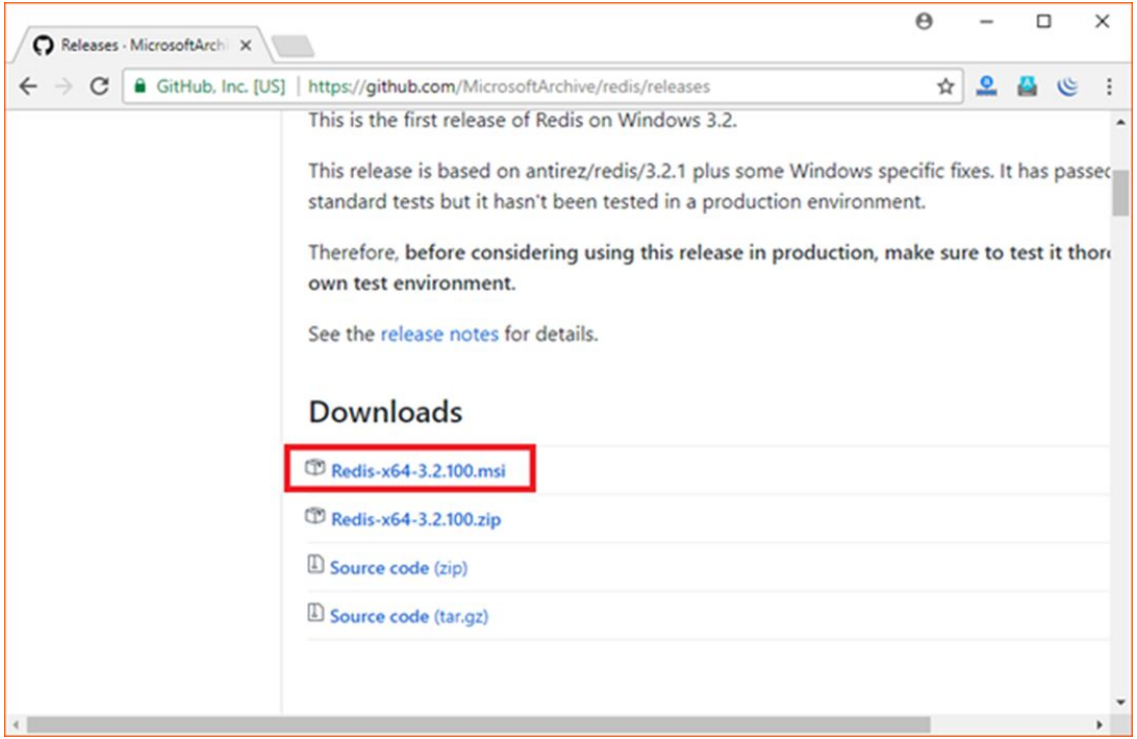
- This is a port for Windows based on Redis.
- We officially support the 64-bit version only. Although you can build the 32-bit version from source if desired
- You can download the latest unsigned binaries and the unsigned MSI installer from the release page.
- For releases prior to 2.8.17.1, the binaries can found in a zip file inside the source archive, under the bin/releas
- Signed binaries are available through NuGet and Chocolatey.
- Redis can be installed as a Windows Service.

Windows-specific changes

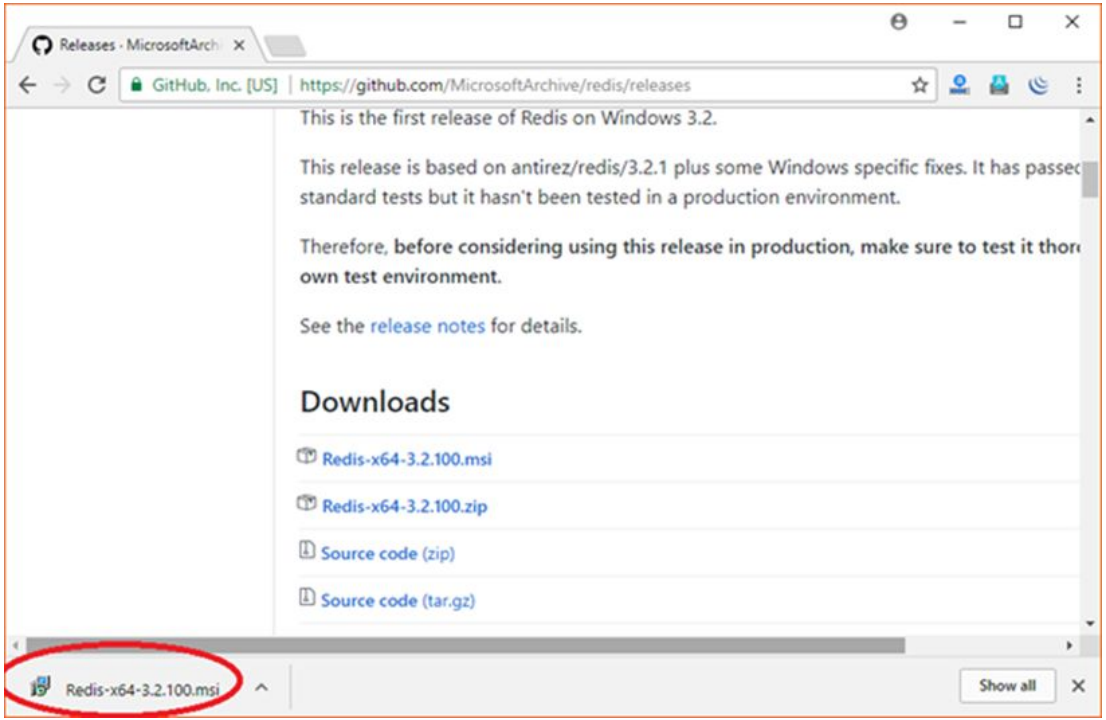
- There is a replacement for the UNIX fork() API that simulates the copy-on-write behavior using a memory ma
  2.8. Version 3.0 is using a similar behavior but dropped the memory mapped file in favor of the system paging

Redis Installation

- Click on download:



Redis Installation

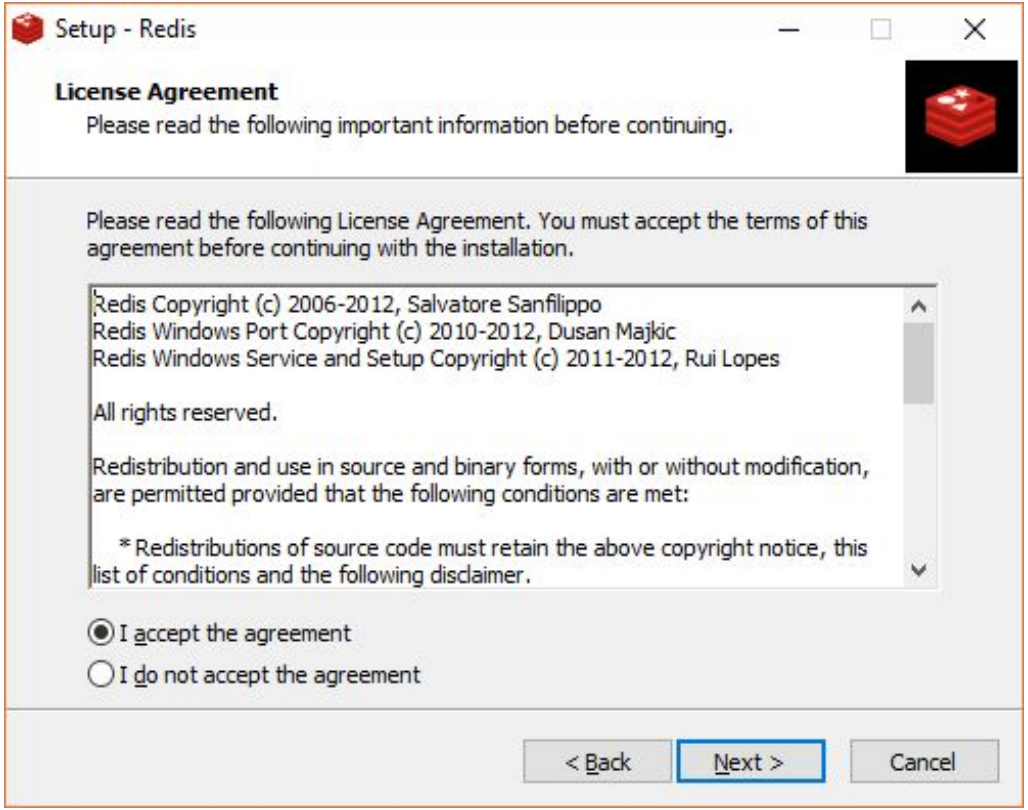


- You can see that Redis is downloaded now.

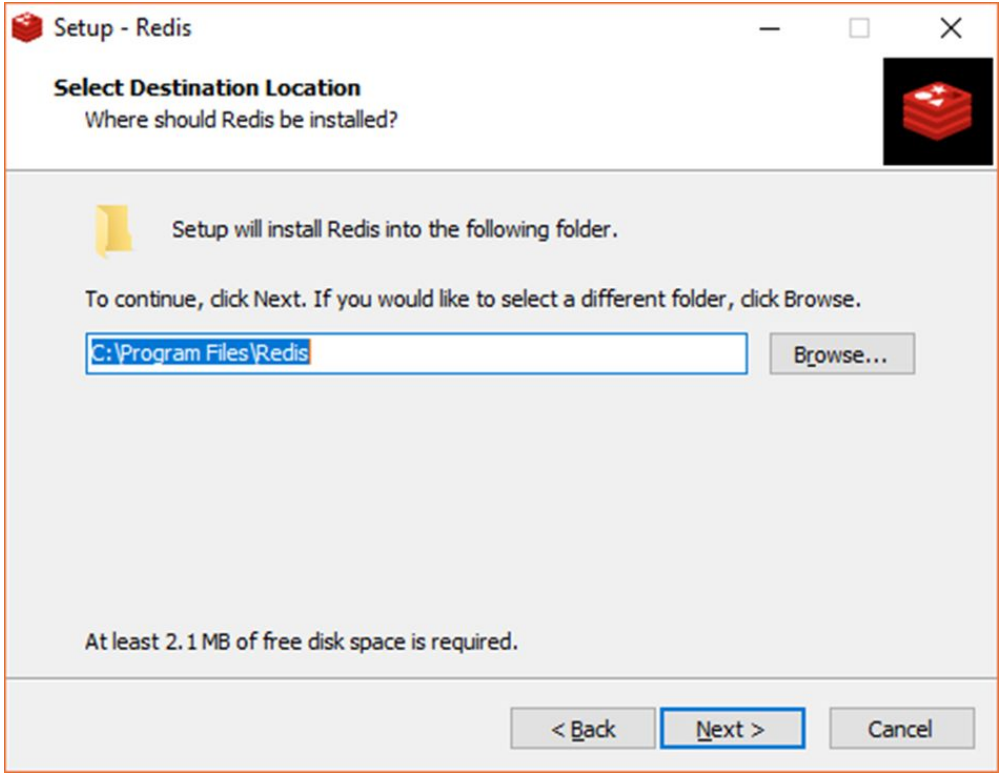
Redis Installation

- You can also download one click Redis and install as a Windows service by using the following Github link.
- https://github.com/rgl/redis/downloads

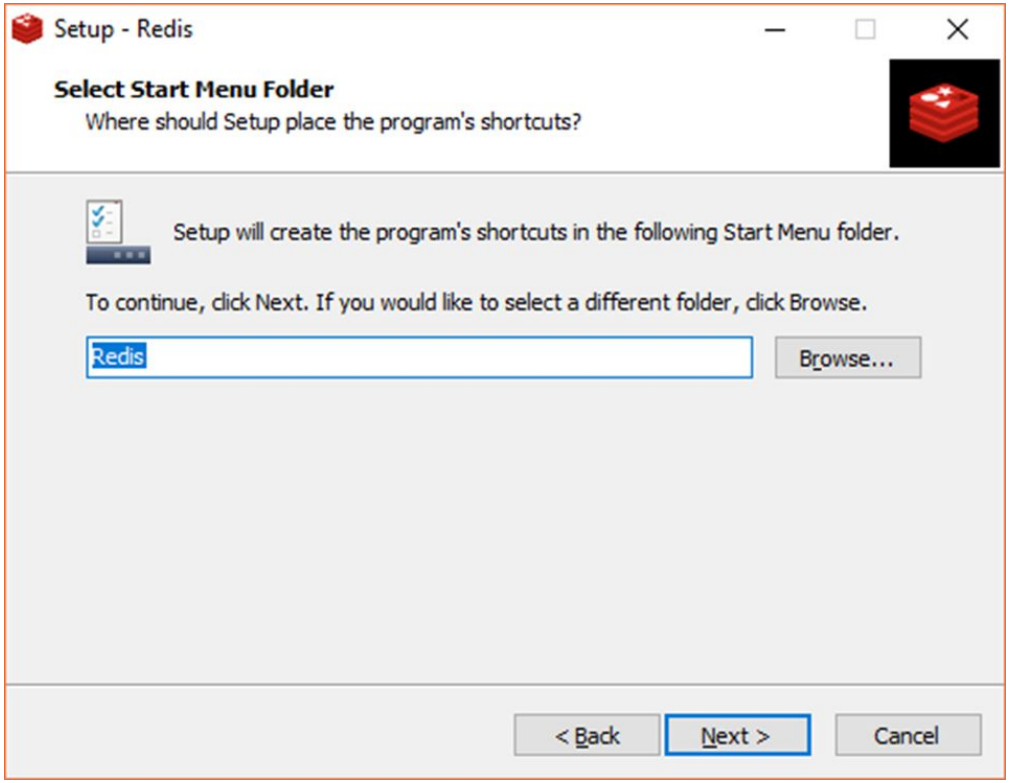
Redis Installation



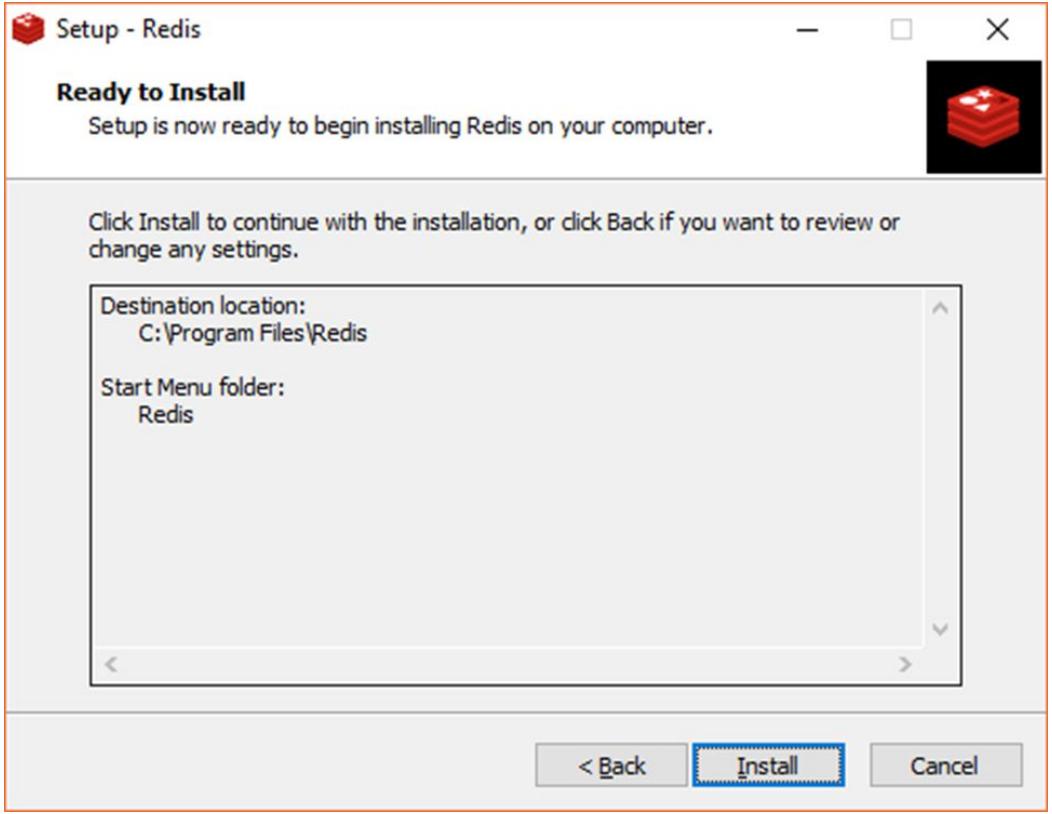
Redis Installation



Redis Installation



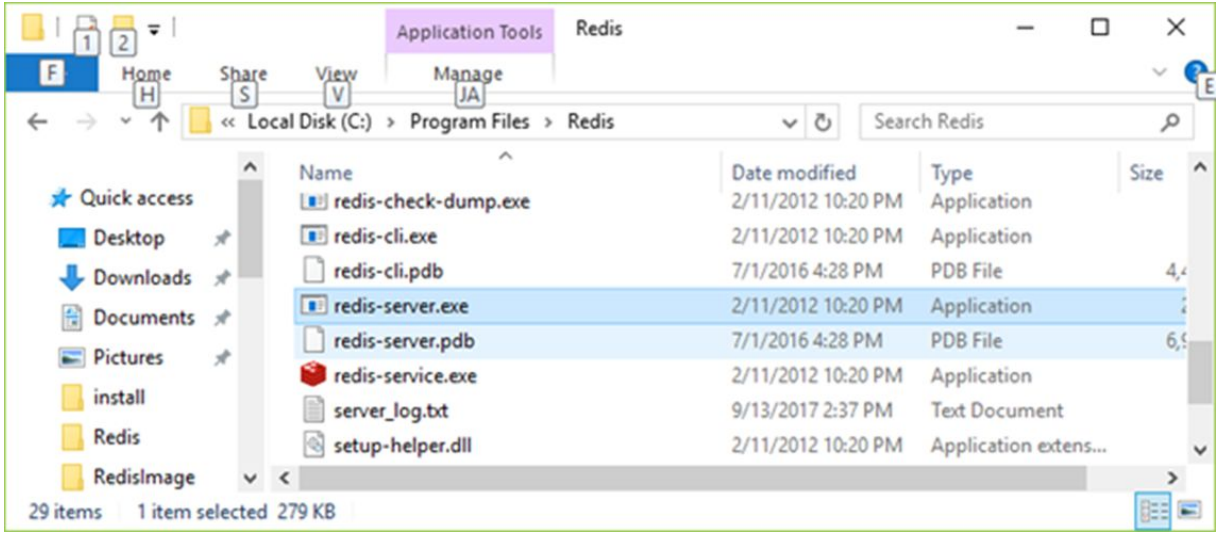
Redis Installation



Redis Installation

Start Redis

- Redis is now ready to use. Start Redis server. Go to program files, followed by redis, followed by:



Start Redis

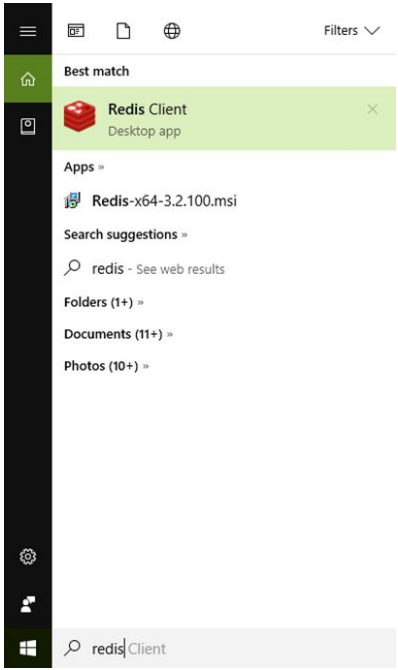
- Click on the redis-server.exe and you will find that the server is started.

C:\Program Files\Redis\redis-server.exe

```
[12912] 24 Sep 14:50:42 # Warning: no config file specified, using the default config. In order to specify a config file
 use 'redis-server /path/to/redis.conf'
[12912] 24 Sep 14:50:42 * Server started, Redis version 2.4.6
[12912] 24 Sep 14:50:42 # Open data file dump.rdb: No such file or directory
[12912] 24 Sep 14:50:42 * The server is now ready to accept connections on port 6379
[12912] 24 Sep 14:50:43 - 0 clients connected (0 slaves), 1179896 bytes in use
[12912] 24 Sep 14:50:48 - 0 clients connected (0 slaves), 1179896 bytes in use
[12912] 24 Sep 14:50:53 - 0 clients connected (0 slaves), 1179896 bytes in use
[12912] 24 Sep 14:50:59 - 0 clients connected (0 slaves), 1179896 bytes in use
```

Start Redis

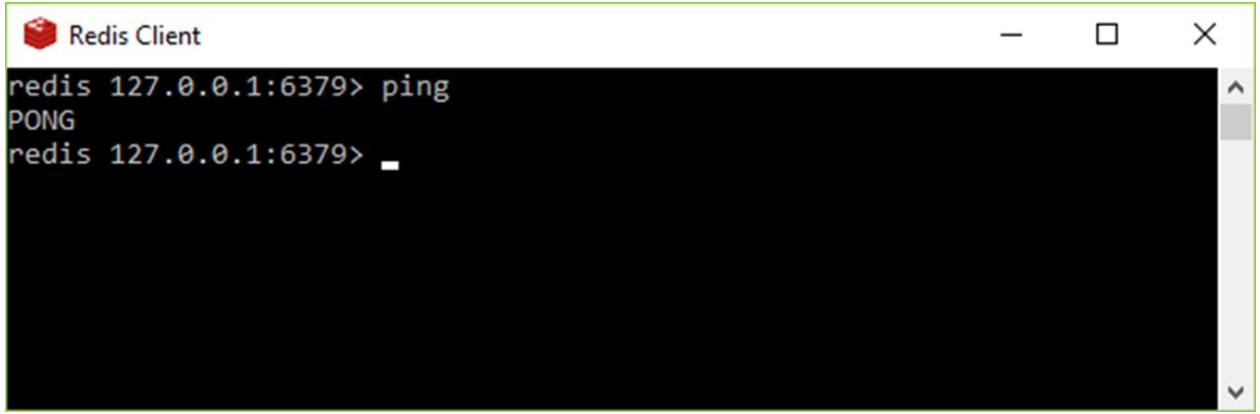
- Now start Redis client.



Start Redis

- Redis is started. Now you can check whether it is connected.

- Use PING command.



Redis Configuration

In Redis, there is a configuration file (redis.conf) available at the root directory of Redis. Although you can get and set all Redis configurations by Redis CONFIG command.

Syntax

Following is the basic syntax of Redis CONFIG command.

```
redis 127.0.0.1:6379> CONFIG GET CONFIG_SETTING_NAME
```

Example

```
redis 127.0.0.1:6379> CONFIG GET loglevel
```

Redis Configuration

Output

Redis Client

```
redis 127.0.0.1:6379> CONFIG GET loglevel
1) "loglevel"
2) "verbose"
redis 127.0.0.1:6379>
```

Edit Configuration

To get all configuration settings, use * in place of CONFIG_SETTING_NAME

Example

```
redis 127.0.0.1:6379> CONFIG GET *
```

Output

Redis Client

```
redis 127.0.0.1:6379> CONFIG GET *
 1) "dir"
 2) "C:\\Program Files\\Redis"
 3) "dbfilename"
 4) "dump.rdb"
 5) "requirepass"
 6) (nil)
 7) "masterauth"
 8) (nil)
 9) "maxmemory"
10) "0"
11) "maxmemory-policy"
12) "volatile-lru"
13) "maxmemory-samples"
14) "3"
15) "timeout"
16) "0"
17) "appendonly"
18) "no"
19) "no-appendfsync-on-rewrite"
20) "no"
21) "appendfsync"
22) "everysec"
23) "save"
24) "3600 1 300 100 60 10000"
25) "auto-aof-rewrite-percentage"
26) "100"
```

Redis Client

```
27) "auto-aof-rewrite-min-size"
28) "1048576"
29) "slave-serve-stale-data"
30) "yes"
31) "hash-max-zipmap-entries"
32) "512"
33) "hash-max-zipmap-value"
34) "64"
35) "list-max-ziplist-entries"
36) "512"
37) "list-max-ziplist-value"
38) "64"
39) "set-max-intset-entries"
40) "512"
41) "zset-max-ziplist-entries"
42) "128"
43) "zset-max-ziplist-value"
44) "64"
45) "slowlog-log-slower-than"
46) "10000"
47) "slowlog-max-len"
48) "64"
49) "loglevel"
50) "verbose"
redis 127.0.0.1:6379>
```

Edit Configuration

To update configuration, you can edit redis.conf file directly or you can update configurations via CONFIG set command.

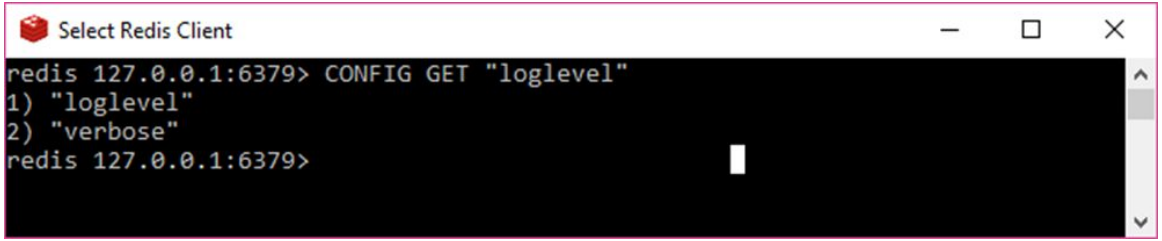
Syntax

Following is the basic syntax of CONFIG SET command.

```
redis 127.0.0.1:6379> CONFIG SET CONFIG_SETTING_NAME NEW_CONFIG_VALUE
```

Example

```
CONFIG GET "loglevel"
```



Redis Data Type

There are five types of data types supported by Redis database.

- Strings
- Hashes
- Lists
- Sets
- Sorted Sets

Strings

- String is a set of bytes. In Redis database, strings are **binary safe**. It means they **have a known length and not determined by any special terminating characters**.
- So it is possible to store anything up to 512 megabytes in one string.

Example

- Let's store a string name "Ajeet Kumar" in the key by using SET command and then retrieve the same by using GET command.

Redis Client

```
redis 127.0.0.1:6379> SET name "Ajeet Kumar"
OK
redis 127.0.0.1:6379> GET name
"Ajeet Kumar"
redis 127.0.0.1:6379>
```

- In the above example, SET and GET are the Redis command, name is the key used in Redis, "Ajeet Kumar" is string value stored in Redis.

Hashes

- Hash is a collection of key-value pairs. In Redis, hashes are maps between string fields and string values. So, they are used to represent objects.

- Let's store a user's object which contains basic information of a user.

Redis Client

```
redis 127.0.0.1:6379> HMSET user:1 username ajeet password javatpoint alexa 2000
OK
redis 127.0.0.1:6379> HGETALL user:1
1) "username"
2) "ajeet"
3) "password"
4) "javatpoint"
5) "alexa"
6) "2000"
```

- Here, HMSET and HGETALL are the command for Redis, while user:1 is the key.

- Every hash can store up to 232 - 1 field-value pairs (more than 4 billion).

Hashes

- Hash is a collection of key-value pairs. In Redis, hashes are maps between string fields and string values. So, they are used to represent objects.

- Let's store a user's object which contains basic information of a user.

Redis Client

```
redis 127.0.0.1:6379> HMSET user:1 username ajeet password javatpoint alexa 2000
OK
redis 127.0.0.1:6379> HGETALL user:1
1) "username"
2) "ajeet"
3) "password"
4) "javatpoint"
5) "alexa"
6) "2000"
```

- Here, HMSET and HGETALL are the command for Redis, while user:1 is the key.

- Every hash can store up to 232 - 1 field-value pairs (more than 4 billion).

Lists

- Redis Lists are defined as a lists of strings, sorted by insertion order. You can add elements to a Redis List on the head or on the tail.

Redis Client

```
redis 127.0.0.1:6379> lpush javatpoint java
(integer) 1
redis 127.0.0.1:6379> lpush javatpoint sql
(integer) 2
redis 127.0.0.1:6379> lpush javatpoint mongodb
(integer) 3
redis 127.0.0.1:6379> lpush javatpoint cassandra
(integer) 4
redis 127.0.0.1:6379> lrange javatpoint 0 10
1) "cassandra"
2) "mongodb"
3) "sql"
4) "java"
redis 127.0.0.1:6379>
```

- The max length of a list is 232 - 1 elements (more than 4 billion of elements per list).

Sets

- Sets are an unordered collection of strings in Redis database. In Redis, you can add, remove, and test for the existence of members in O(1) time complexity.

Redis Client

```
redis 127.0.0.1:6379> sadd tutoriallist redis
(integer) 1
redis 127.0.0.1:6379> sadd tutoriallist sql
(integer) 1
redis 127.0.0.1:6379> sadd tutoriallist postgresql
(integer) 1
redis 127.0.0.1:6379> sadd tutoriallist postgresql
(integer) 0
redis 127.0.0.1:6379> sadd tutoriallist postgresql
(integer) 0
redis 127.0.0.1:6379> smembers tutoriallist
1) "redis"
2) "postgresql"
3) "sql"
redis 127.0.0.1:6379>
```

- In the above example, you can see that postgresql is added thrice but due to unique property of the set it is added only once.

- The max number of members in a set is 232 - 1 elements (more than 4 billion of elements per list).

Sorted Sets

- Redis Sorted Sets are similar to Redis Sets. They are also a set of non-repeating collections of Strings.

- But every member of a Sorted Set is associated with a score, that is used in order to take the sorted set ordered, from the smallest to the greatest score. While members are unique, the scores may be repeated.

Redis Client

```
redis 127.0.0.1:6379> zadd tutoriallist 0 redis
(integer) 1
redis 127.0.0.1:6379> zadd tutoriallist 0 sql
(integer) 1
redis 127.0.0.1:6379> zadd tutoriallist 0 postgresql
(integer) 1
redis 127.0.0.1:6379> zadd tutoriallist 0 postgresql
(integer) 0
redis 127.0.0.1:6379> zadd tutoriallist 0 postgresql
(integer) 0
redis 127.0.0.1:6379> ZRANGEBYSCORE tutoriallist 0 10
1) "postgresql"
2) "redis"
3) "sql"
redis 127.0.0.1:6379>
```

Redis – Java - Installation

Now, let us see how to set up Redis Java driver.

- You need to download the jar from the path **Download jedis.jar**. Make sure to download the latest release of it.
- You need to include the **jedis.jar** into your classpath.

Connect to Redis Server

```
import redis.clients.jedis.Jedis;

public class RedisJava {
   public static void main(String[] args) {
      //Connecting to Redis server on localhost
      Jedis jedis = new Jedis("localhost");
      System.out.println("Connection to server sucessfully");
      //check whether server is running or not
      System.out.println("Server is running: "+jedis.ping());
   }
}
```

Connect to Redis Server

- Now, let's compile and run the above program to test the connection to Redis server. You can change your path as per your requirement.
- We are assuming the current version of **jedis.jar** is available in the current path.

```
$javac RedisJava.java
$java RedisJava
Connection to server sucessfully
Server is running: PONG
```

Redis Java String Example

```
import redis.clients.jedis.Jedis;

public class RedisStringJava {
   public static void main(String[] args) {
      //Connecting to Redis server on localhost
      Jedis jedis = new Jedis("localhost");
      System.out.println("Connection to server sucessfully");
      //set the data in redis string
      jedis.set("tutorial-name", "Redis tutorial");
      // Get the stored data and print it
      System.out.println("Stored string in redis:: "+ jedis.get("tutorial-name"));
   }
}
```

Redis Java String Example

- Now, let's compile and run the above program.

```
$javac RedisStringJava.java
$java RedisStringJava
Connection to server sucessfully
Stored string in redis:: Redis tutorial
```

Redis Java List Example

```
import redis.clients.jedis.Jedis;

public class RedisListJava {
   public static void main(String[] args) {
      //Connecting to Redis server on localhost
      Jedis jedis = new Jedis("localhost");
      System.out.println("Connection to server sucessfully");
      //store data in redis list
      jedis.lpush("tutorial-list", "Redis");
      jedis.lpush("tutorial-list", "Mongodb");
      jedis.lpush("tutorial-list", "Mysql");
      // Get the stored data and print it
      List<String> list = jedis.lrange("tutorial-list", 0 ,5);
      for(int i = 0; i<list.size(); i++) {
         System.out.println("Stored string in redis:: "+list.get(i));
      }
   }
}
```

Redis Java List Example

- Now, let's compile and run the above program.

```
$javac RedisListJava.java
$java RedisListJava
Connection to server sucessfully
Stored string in redis:: Redis
Stored string in redis:: Mongodb
Stored string in redis:: Mysql
```

Redis Java Keys Example

```
import redis.clients.jedis.Jedis;

public class RedisKeyJava {
   public static void main(String[] args) {

      //Connecting to Redis server on localhost
      Jedis jedis = new Jedis("localhost");
      System.out.println("Connection to server sucessfully");
      //store data in redis list
      // Get the stored data and print it
      List<String> list = jedis.keys("*");

      for(int i = 0; i<list.size(); i++) {
         System.out.println("List of stored keys:: "+list.get(i));
      }
   }
}
```

Redis Java Keys Example

- Now, let's compile and run the above program.

```
$javac RedisKeyJava.java
$java RedisKeyJava
Connection to server sucessfully
List of stored keys:: tutorial-name
List of stored keys:: tutorial-list
```

Messaging with Redis

- You will build an application that uses StringRedisTemplate to publish a string message and has a POJO subscribe for the message by using MessageListenerAdapter.
- It may sound strange to be using Spring Data Redis as the means to publish messages, but, as you will discover, Redis provides not only a NoSQL data store but a messaging system as well.

What You Need

- JDK 1.8 or later
- Maven 3.2+
- You can also import the code straight into your IDE:
  - Spring Tool Suite (STS)
  - IntelliJ IDEA
- A Redis server

Starting with Spring Initializr

- For all Spring applications, you should start with the Spring Initializr.
- The Initializr offers a fast way to pull in all the dependencies you need for an application and does a lot of the set up for you.
- This example needs only the Spring for Redis dependency. The following image shows the Initializr set up for this sample project:

Starting with Spring Initializr

Spring Initializr
Bootstrap your application

Light UI | Github | Twitter | Help

Project: Maven Project | Gradle Project

Language: Java | Kotlin | Groovy

Spring Boot: 2.2.1 (SNAPSHOT) | 2.2.0 | 2.1.10 (SNAPSHOT) | 2.1.9

Project Metadata

Group
com.example

Artifact
messaging-redis

> Options

Dependencies

1 selected

Search dependencies to add
Web, Security, JPA, Actuator, Devtools...

Selected dependencies

Spring Data Redis (Access+Driver)
Advanced and thread-safe Java Redis client for synchronous, asynchronous, and reactive usage. Supports Cluster, Sentinel, Pipelining, Auto-Reconnect, Codecs and much more.

Generate - ⌘ + ↵ | Explore - Ctrl + Space | Share...

Starting with Spring Initializr

- The following listing shows the pom.xml file that is created when you choose Maven:

pom_redis.xml

Create a Redis Message Receiver

- In any messaging-based application, there are message publishers and messaging receivers.
- To create the message receiver, implement a receiver with a method to respond to messages, as the following example (from src/main/java/com/example/messagingredis/Receiver.java) shows:

```
package com.example.messagingredis;
import java.util.concurrent.atomic.AtomicInteger;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
public class Receiver {
    private static final Logger LOGGER = LoggerFactory.getLogger(Receiver.class);
    private AtomicInteger counter = new AtomicInteger();
    public void receiveMessage(String message) {
        LOGGER.info("Received <" + message + ">");
        counter.incrementAndGet();
    }
    public int getCount() {
        return counter.get();
    }
}
```

Create a Redis Message Receiver

- The Receiver is a POJO that defines a method for receiving messages.
- When you register the Receiver as a message listener, you can name the message-handling method whatever you want.
- For demonstration purposes, the receiver is counting the messages received. That way, it can signal when it has received a message.

Register the Listener and Send a Message

- Spring Data Redis provides all the components you need to send and receive messages with Redis.
- Specifically, you need to configure:
  - A connection factory
  - A message listener container
  - A Redis template

Register the Listener and Send a Message

- You will use the Redis template to send messages, and you will register the Receiver with the message listener container so that it will receive messages.
- The connection factory drives both the template and the message listener container, letting them connect to the Redis server.
- This example uses Spring Boot's default RedisConnectionFactory, an instance of JedisConnectionFactory that is based on the Jedis Redis library.

Register the Listener and Send a Message

- The connection factory is injected into both the message listener container and the Redis template, as the following example (from src/main/java/com/example/messagingredis/MessagingRedisApplication.java) shows:

MessagingRedisApplication.java

Register the Listener and Send a Message

- The bean defined in the listenerAdapter method is registered as a message listener in the message listener container defined in container and will listen for messages on the chat topic.
- Because the Receiver class is a POJO, it needs to be wrapped in a message listener adapter that implements the MessageListener interface (which is required by addMessageListener()).
- The message listener adapter is also configured to call the receiveMessage() method on Receiver when a message arrives.

Register the Listener and Send a Message

- The connection factory and message listener container beans are all you need to listen for messages.
- To send a message, you also need a Redis template.
- Here, it is a bean configured as a StringRedisTemplate, an implementation of RedisTemplate that is focused on the common use of Redis, where both keys and values are String instances.
- The main() method kicks off everything by creating a Spring application context.

Register the Listener and Send a Message

- The application context then starts the message listener container, and the message listener container bean starts listening for messages.
- The main() method then retrieves the StringRedisTemplate bean from the application context and uses it to send a Hello from Redis! message on the chat topic.
- Finally, it closes the Spring application context, and the application ends.

Build an executable JAR

- You can run the application from the command line with Gradle or Maven. You can also build a single executable JAR file that contains all the necessary dependencies, classes, and resources and run that.
- Building an executable jar makes it easy to ship, version, and deploy the service as an application throughout the development lifecycle, across different environments, and so forth.
- If you use Maven, you can run the application by using ./mvnw spring-boot:run. Alternatively, you can build the JAR file with ./mvnw clean package and then run the JAR file, as follows:

```
java -jar target/gs-messaging-redis-0.1.0.jar
```

Build an executable JAR

- You can run the application from the command line with Gradle or Maven. You can also build a single executable JAR file that contains all the necessary dependencies, classes, and resources and run that.
- Building an executable jar makes it easy to ship, version, and deploy the service as an application throughout the development lifecycle, across different environments, and so forth.
- If you use Maven, you can run the application by using ./mvnw spring-boot:run. Alternatively, you can build the JAR file with ./mvnw clean package and then run the JAR file, as follows:

```
java -jar target/gs-messaging-redis-0.1.0.jar
```

- The steps described here create a runnable JAR. You can also build a classic WAR file.

Build an executable JAR

You should see output similar to the following:

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v2.1.8.RELEASE)

2019-09-23 12:57:11.578  INFO 35396 --- [           main] c.e.m.MessagingRedisApplication          : Starting MessagingRedisApplication on Jays-MBP with PID 35396
(/Users/j/projects/guides/gs-messaging-redis/complete/target/classes started by j in /Users/j/projects/guides/gs-messaging-redis/complete)
2019-09-23 12:57:11.581  INFO 35396 --- [           main] c.e.m.MessagingRedisApplication          : No active profile set, falling back to default profiles:
default
2019-09-23 12:57:11.885  INFO 35396 --- [           main] .s.d.r.c.RepositoryConfigurationDelegate : Multiple Spring Data modules found, entering strict
repository configuration mode!
2019-09-23 12:57:11.887  INFO 35396 --- [           main] .s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data repositories in DEFAULT mode.
2019-09-23 12:57:11.914  INFO 35396 --- [           main] .s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 13ms. Found 0
repository interfaces.
2019-09-23 12:57:12.685  INFO 35396 --- [    container-1] io.lettuce.core.EpollProvider            : Starting without optional epoll library
2019-09-23 12:57:12.685  INFO 35396 --- [    container-1] io.lettuce.core.KqueueProvider           : Starting without optional kqueue library
2019-09-23 12:57:12.848  INFO 35396 --- [           main] c.e.m.MessagingRedisApplication          : Started MessagingRedisApplication in 1.511 seconds (JVM
running for 3.685)
2019-09-23 12:57:12.849  INFO 35396 --- [           main] c.e.m.MessagingRedisApplication          : Sending message...
2019-09-23 12:57:12.861  INFO 35396 --- [    container-2] com.example.messagingredis.Receiver      : Received <Hello from Redis!>
```

TIA-Academy

Springboot Redis

Springboot Redis

- Let's declare the necessary dependencies in our pom.xml for the example application we are building:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Redis Configuration

- We need to connect our application with the Redis server. To establish this connection, we are using Jedis, a Redis client implementation.
- Let's start with the configuration bean definitions:

```
@Bean
JedisConnectionFactory jedisConnectionFactory() {
    return new JedisConnectionFactory();
}
@Bean
public RedisTemplate<String, Object> redisTemplate() {
    final RedisTemplate<String, Object> template = new RedisTemplate<String, Object>();
    template.setConnectionFactory(jedisConnectionFactory());
    template.setValueSerializer(new GenericToStringSerializer<Object>(Object.class));
    return template;
}
```

- The JedisConnectionFactory is made into a bean so that we can create a RedisTemplate to query data.

Message Publisher

- Following the principles of SOLID, we create a MessagePublisher interface:

```
public interface MessagePublisher {
    void publish(final String message);
}
```

Message Publisher

- We implement the MessagePublisher interface to use the high-level RedisTemplate to publish the message since the RedisTemplate allows arbitrary objects to be passed in as messages:

```
@Service
public class MessagePublisherImpl implements MessagePublisher {
    @Autowired
    private RedisTemplate<String, Object> redisTemplate;
    @Autowired
    private ChannelTopic topic;
    public MessagePublisherImpl() {
    }
    public MessagePublisherImpl(final RedisTemplate<String, Object> redisTemplate, final ChannelTopic topic) {
        this.redisTemplate = redisTemplate;
        this.topic = topic;
    }
    public void publish(final String message) {
        redisTemplate.convertAndSend(topic.getTopic(), message);
    }
}
```

Message Publisher

- We also define this as a bean in RedisConfig:

```
@Bean
MessagePublisher redisPublisher() {
    return new MessagePublisherImpl(redisTemplate(), topic());
}
```

Message Listener

- In order to subscribe to messages, we need to implement the MessageListener interface: each time a new message arrives, a callback gets invoked and the user code executed through a method named onMessage.
- This interface gives access to the message, the channel it has been received through, and any pattern used by the subscription to match the channel.

Message Listener

- Thus, we create a service class to implement MessageSubscriber:

```
@Service
public class MessageSubscriber implements MessageListener {
    public static List<String> messageList = new ArrayList<String>();
    public void onMessage(final Message message, final byte[] pattern) {
        messageList.add(message.toString());
        System.out.println("Message received: " + new String(message.getBody()));
    }
}
```

- We add a bean definition to RedisConfig:

```
@Bean
MessageListenerAdapter messageListener() {
    return new MessageListenerAdapter(new MessageSubscriber());
}
```

RedisRepository

- Now that we have configured the application to interact with the Redis server, we are going to prepare the application to take example data.

Model

- For this example, we are defining a Movie model with two fields:

```
private String id;
private String name;
//standard getters and setters
```

Repository interface

- Unlike other Spring Data projects, Spring Data Redis does offer any features to build on top of the other Spring Data interfaces. This is odd for us who have experience with the other Spring Data projects.
- Often, there is no need to write an implementation of a repository interface with Spring Data projects.
- We simply just interact with the interface. Spring Data JPA provides numerous repository interfaces that can be extended to get features such as CRUD operations, derived queries, and paging.

Repository interface

- So, unfortunately, we need to write our own interface and then define the methods:

```
public interface RedisRepository {
    Map<Object, Object> findAllMovies();
    void add(Movie movie);
    void delete(String id);
    Movie findMovie(String id);
}
```

Repository Implementation

- Our implementation class uses the redisTemplate defined in our configuration class RedisConfig.
- We use the HashOperations template that Spring Data Redis offers:

RedisRepositoryImpl.java

- Let’s take note of the init() method. In this method, we use a function named opsForHash(), which returns the operations performed on hash values bound to the given key.
- We then use the hashOps, which was defined in init(), for all of our CRUD operations.

Web Interface

- In this section, we will review adding Redis CRUD operations capabilities to a web interface.

Add a Movie

- We want to be able to add a Movie to our web page.
- The Key is the is the Movie id and the Value is the actual object.
- However, we will later address this, so only the Movie name is shown as the value.

Web Interface

- Let’s add a form to an HTML document and assign appropriate names and IDs:

```
<form id="addForm">
<div class="form-group">
      <label for="keyInput">Movie ID (key)</label>
      <input name="keyInput" id="keyInput" class="form-control"/>
</div>
<div class="form-group">
      <label for="valueInput">Movie Name (field of Movie object value)</label>
      <input name="valueInput" id="valueInput" class="form-control"/>
</div>
<button class="btn btn-default" id="addButton">Add</button>
</form>
```

Web Interface

- Now, we use JavaScript to persist the values on form submission:

```
$(document).ready(function() {
    var keyInput = $('#keyInput'),
        valueInput = $('#valueInput');
    refreshTable();
    $('#addForm').on('submit', function(event) {
        var data = {
            key: keyInput.val(),
            value: valueInput.val()
        };
        $.post('/add', data, function() {
            refreshTable();
            keyInput.val('');
            valueInput.val('');
            keyInput.focus();
        });
        event.preventDefault();
    });
    keyInput.focus();
});
```

Web Interface

- We assign the @RequestMapping value for the POST request, request the Key and Value, create a Movie object, and save it to the repository:

```
@RequestMapping(value = "/add", method = RequestMethod.POST)
public ResponseEntity<String> add(
    @RequestParam String key,
    @RequestParam String value) {
    Movie movie = new Movie(key, value);
    redisRepository.add(movie);
    return new ResponseEntity<>(HttpStatus.OK);
}
```

Viewing the Content

- Once a Movie object is added, we refresh the table to display an updated table.
- In our JavaScript code block for section 7.1, we called a JavaScript function called refreshTable().
- This function performs a GET request to retrieve the current data in the repository:

```
function refreshTable() {
    $.get('/values', function(data) {
        var attr,
            mainTable = $('#mainTable tbody');
        mainTable.empty();
        for (attr in data) {
            if (data.hasOwnProperty(attr)) {
                mainTable.append(row(attr, data[attr]));
            }
        }
    });
}
```

Viewing the Content

- The GET request is processed by a method named findAll() that retrieves all the Movie objects stored in the repository and then converts the datatype from Map<Object, Object> to Map<String, String>:

```
@RequestMapping("/values")
public @ResponseBody Map<String, String> findAll() {
    Map<Object, Object> aa = redisRepository.findAllMovies();
    Map<String, String> map = new HashMap<String, String>();
    for(Map.Entry<Object, Object> entry : aa.entrySet()){
        String key = (String) entry.getKey();
        map.put(key, aa.get(key).toString());
    }
    return map;
}
```

Delete a Movie

- We write JavaScript to do a POST request to /delete, refresh the table, and set keyboard focus to key input:

```
function deleteKey(key) {
    $.post('/delete', {key: key}, function() {
        refreshTable();
        $('#keyInput').focus();
    });
}
```

Delete a Movie

- We request the Key and delete the object in the redisRepository based on this key:

```
@RequestMapping(value = "/delete", method = RequestMethod.POST)
public ResponseEntity<String> delete(@RequestParam String key) {
    redisRepository.delete(key);
    return new ResponseEntity<>(HttpStatus.OK);
}
```

Demo

- Here, we added two movies:

Movie ID (key)

Movie Name (field of Movie object value)

Add

Key	Value		
key B	Alien	Edit	Delete
key A	Gone with the Wind	Edit	Delete

Demo

- Here, we removed one movie:

Movie ID (key)

Movie Name (field of Movie object value)

Add

Key	Value		
key B	Alien	Edit	Delete

TIA-Academy

ASSIGNMENT 05
(HOME ASSIGNMENT)

Sources

- https://www.javatpoint.com/redis-tutorial
- https://spring.io/guides/gs/messaging-redis/
- https://www.javacodegeeks.com/2017/11/intro-redis-spring-boot.html

TIA-Academy

Thank You