Lettuce Reference Guide

Mark Paluch

5.1.0.M1

Table of Contents

Chapter 1. Overview

This document is the reference guide for Lettuce. It explains how to use Lettuce, its concepts, semantics, and the syntax.

You can read this reference guide in a linear fashion, or you can skip sections if something does not interest you.

This section provides some basic introduction to Redis. The rest of the document refers only to Lettuce features and assumes the user is familiar with Redis concepts.

1.1. Knowing Redis

NoSQL stores have taken the storage world by storm. It is a vast domain with a plethora of solutions, terms and patterns (to make things worse even the term itself has multiple meanings). While some of the principles are common, it is crucial that the user is familiar to some degree with Redis. The best way to get acquainted to this solutions is to read their documentation and follow their documentation - it usually doesn't take more then 5-10 minutes to go through them and if you are coming from an RDMBS-only background many times these exercises can be an eye opener.

The jumping off ground for learning about Redis is redis.io. Here is a list of other useful resources:

- The interactive tutorial introduces Redis.
- The command references explains Redis commands and contains links to getting started guides, reference documentation and tutorials.

1.2. Project Reactor

Reactor is a highly optimized reactive library for building efficient, non-blocking applications on the JVM based on the Reactive Streams Specification. Reactor based applications can sustain very high throughput message rates and operate with a very low memory footprint, making it suitable for building efficient event-driven applications using the microservices architecture.

Reactor implements two publishers Flux<T> and Mono<T>, both of which support non-blocking back-pressure. This enables exchange of data between threads with well-defined memory usage, avoiding unnecessary intermediate buffering or blocking.

1.3. Non-blocking API for Redis

Lettuce is a scalable thread-safe Redis client based on netty and Reactor. Lettuce provides synchronous, asynchronous and reactive APIs to interact with Redis.

1.4. Requirements

Lettuce 4.x and 5.x binaries requires JDK level 8.0 and above.

In terms of Redis, at least 2.6.

1.5. Additional Help Resources

Learning a new framework is not always straight forward. In this section, we try to provide what we think is an easy to follow guide for starting with Lettuce. However, if you encounter issues or you are just looking for an advice, feel free to use one of the links below:

1.5.1. Support

There are a few support options available:

- Lettuce on Stackoverflow Stackoverflow is a tag for all Lettuce users to share information and help each other. Note that registration is needed **only** for posting.
- Get in touch with the community on Gitter.
- Google Group: lettuce-redis-client-users or lettuce-redis-client-users@googlegroups.com.
- Report bugs (or ask questions) in Github issues https://github.com/lettuce-io/lettuce-core/issues.

1.5.2. Following Development

For information on the Lettuce source code repository, nightly builds and snapshot artifacts please see the Lettuce homepage. You can help make lettuce best serve the needs of the lettuce community by interacting with developers through the Community on Stackoverflow. To follow developer activity look for the mailing list information on the lettuce homepage. If you encounter a bug or want to suggest an improvement, please create a ticket on the lettuce issue tracker.

1.5.3. Project Metadata

- Version Control – https://github.com/lettuce-io/lettuce-core
- Releases and Binary Packages – https://github.com/lettuce-io/lettuce-core/releases
- Issue tracker – https://github.com/lettuce-io/lettuce-core/issues
- Release repository – https://repo1.maven.org/maven2/ (Maven Central)
- Snapshot repository – https://oss.sonatype.org/content/repositories/snapshots/ (OSS Sonatype Snapshots)

1.6. Where to go from here

- Head to Getting Started if you feel like jumping straight into the code.
- Go to High-Availability and Sharding for Master/Slave, Redis Sentinel and Redis Cluster topics.
- In order to dig deeper into the core features of Reactor:
  - If you're looking for client configuration options, performance related behavior and how to use various transports, go to Advanced usage.
  - See Integration and Extension for extending Lettuce with codecs or integrate it in your CDI/Spring application.
  - You want to know more about **at-least-once** and **at-most-once**? Take a look into Command

execution reliability.

Chapter 2. New & Noteworthy

2.1. What's new in Lettuce 5.0

- New artifact coordinates: `io.lettuce:lettuce-core` and packages moved from `com.lambdaworks.redis` to `io.lettuce.core`.
- Reactive API now Reactive Streams-based using Project Reactor.
- Redis Command Interfaces supporting dynamic command invocation and Redis Modules.
- Enhanced, immutable Key-Value objects.
- Asynchronous Cluster connect.
- Native transport support for Kqueue on macOS systems.
- Removal of support for Guava.
- Removal of deprecated `RedisConnection` and `RedisAsyncConnection` interfaces.
- Java 9 compatibility.
- HTML and PDF reference documentation along with a new project website: https://lettuce.io.

Chapter 3. Getting Started

You can get started with lettuce in various ways.

3.1. 1. Get it

3.1.1. For Maven users:

Add these lines to file pom.xml:

```
<dependency>
    <groupId>io.lettuce</groupId>
    <artifactId>lettuce-core</artifactId>
    <version>5.1.0.M1</version>
</dependency>
```

3.1.2. For Ivy users:

Add these lines to file ivy.xml:

```
<ivy-module>
  <dependencies>
    <dependency org="io.lettuce" name="lettuce-core" rev="5.1.0.M1"/>
  </dependencies>
</ivy-module>
```

3.1.3. For Gradle users:

Add these lines to file build.gradle:

```
dependencies {
  compile 'io.lettuce:lettuce-core:5.1.0.M1'
}
```

3.1.4. Plain Java

Download the latest binary package from https://github.com/lettuce-io/lettuce-core/releases and extract the archive.

3.2. 2. Start coding

So easy! No more boring routines, we can start.

Import required classes:

```
import io.lettuce.core.*;
```

and now, write your code:

```
RedisClient redisClient = RedisClient.create("redis://password@localhost:6379/0");
StatefulRedisConnection<String, String> connection = redisClient.connect();
RedisCommands<String, String> syncCommands = connection.sync();

syncCommands.set("key", "Hello, Redis!");

connection.close();
redisClient.shutdown();
```

Done!

Do you want to see working examples?

- Standalone Redis
- Standalone Redis with SSL
- Redis Sentinel
- Redis Cluster
- Connecting to a ElastiCache Master
- Connecting to ElastiCache with Master/Slave
- Connecting to Azure Redis Cluster
- Lettuce with Spring

Chapter 4. Connecting Redis

Connections to a Redis Standalone, Sentinel, or Cluster require a specification of the connection details. The unified form is `RedisURI`. You can provide the database, password and timeouts within the `RedisURI`. You have following possibilities to create a `RedisURI`:

1. Use an URI:

```
RedisURI.create("redis://localhost/");
```

2. Use the Builder

```
RedisURI.Builder.redis("localhost", 6379).auth("password").database(1).build();
```

3. Set directly the values in `RedisURI`

```
new RedisURI("localhost", 6379, 60, TimeUnit.SECONDS);
```

4.1. URI syntax

Redis Standalone

redis **://** [**:** *password@*] *host* [**:** *port*] [**/** *database*][**?** [*timeout*=*timeout*[*d*|*h*|*m*|*s*|*ms*|*us*|*ns*]] [&_database=database_]]

Redis Standalone (SSL)

rediss **://** [**:** *password@*] *host* [**:** *port*] [**/** *database*][**?** [*timeout*=*timeout*[*d*|*h*|*m*|*s*|*ms*|*us*|*ns*]] [&_database=database_]]

Redis Standalone (Unix Domain Sockets)

redis-socket **://** *path* [**?**[*timeout*=*timeout*[*d*|*h*|*m*|*s*|*ms*|*us*|*ns*]][&_database=database_]]

Redis Sentinel

redis-sentinel **://** [**:** *password@*] *host1*[**:** *port1*] [**,** *host2*[**:** *port2*]] [**,** *hostN*[**:** *portN*]] [**/** *database*][**?** [*timeout*=*timeout*[*d*|*h*|*m*|*s*|*ms*|*us*|*ns*]] [&_sentinelMasterId=sentinelMasterId_] [&_database=database_]]

Schemes

- `redis` Redis Standalone
- `rediss` Redis Standalone SSL
- `redis-socket` Redis Standalone Unix Domain Socket

- `redis-sentinel` Redis Sentinel

Timeout units

- `d` Days
- `h` Hours
- `m` Minutes
- `s` Seconds
- `ms` Milliseconds
- `us` Microseconds
- `ns` Nanoseconds

Hint: The database parameter within the query part has higher precedence than the database in the path.

RedisURI supports Redis Standalone, Redis Sentinel and Redis Cluster with plain, SSL, TLS and unix domain socket connections.

4.2. Basic Usage

Example 1. Basic usage

```
RedisClient client = RedisClient.create("redis://localhost");          ①

StatefulRedisConnection<String, String> connection = client.connect(); ②

RedisCommands<String, String> commands = connection.sync();            ③

String value = commands.get("foo");                                    ④

...

connection.close();                                                    ⑤

client.shutdown();                                                     ⑥
```

① Create the `RedisClient` instance and provide a Redis URI pointing to localhost, Port 6379 (default port).

② Open a Redis Standalone connection. The endpoint is used from the initialized `RedisClient`

③ Obtain the command API for synchronous execution. Lettuce supports asynchronous and reactive execution models, too.

④ Issue a `GET` command to get the key `foo`.

⑤ Close the connection when you're done. This happens usually at the very end of your application. Connections are designed to be long-lived.

⑥ Shut down the client instance to free threads and resources. This happens usually at the very end of your application.

Each Redis command is implemented by one or more methods with names identical to the lowercase Redis command name. Complex commands with multiple modifiers that change the result type include the CamelCased modifier as part of the command name, e.g. `zrangebyscore` and `zrangebyscoreWithScores`.

Redis connections are designed to be long-lived and thread-safe, and if the connection is lost will reconnect until `close()` is called. Pending commands that have not timed out will be (re)sent after successful reconnection.

All connections inherit a default timeout from their RedisClient and
and will throw a `RedisException` when non-blocking commands fail to return a result before the timeout expires. The timeout defaults to 60 seconds and may be changed in the RedisClient or for each connection. Synchronous methods will throw a `RedisCommandExecutionException` in case Redis responds with an error. Asynchronous connections do not throw exceptions when Redis responds with an error.

4.2.1. RedisURI

The RedisURI contains the host/port and can carry authentication/database details. On a successful

connect you get authenticated, and the database is selected afterward. This applies also after re-establishing a connection after a connection loss.

A Redis URI can also be created from an URI string. Supported formats are:

- `redis://[password@]host[:port][/databaseNumber]` Plaintext Redis connection
- `rediss://[password@]host[:port][/databaseNumber]` SSL Connections Redis connection
- `redis-sentinel://[password@]host[:port][,host2[:port2]][/databaseNumber]#sentinelMasterId` for using Redis Sentinel
- `redis-socket:///path/to/socket` Unix Domain Sockets connection to Redis

4.2.2. Exceptions

In the case of an exception/error response from Redis, you'll receive a `RedisException` containing the error message. `RedisException` is a `RuntimeException`.

4.2.3. Examples

Example 2. Using host and port and set the default timeout to 20 seconds

```
RedisClient client = RedisClient.create(RedisURI.create("localhost", 6379));
client.setDefaultTimeout(20, TimeUnit.SECONDS);

// ...

client.shutdown();
```

Example 3. Using RedisURI

```
RedisURI redisUri = RedisURI.Builder.redis("localhost")
                                .withPassword("authentication")
                                .withDatabase(2)
                                .build();
RedisClient client = RedisClient.create(rediUri);

// ...

client.shutdown();
```

Example 4. SSL RedisURI

```
RedisURI redisUri = RedisURI.Builder.redis("localhost")
                                    .withSsl(true)
                                    .withPassword("authentication")
                                    .withDatabase(2)
                                    .build();
RedisClient client = RedisClient.create(redisUri);

// ···

client.shutdown();
```

Example 5. String RedisURI

```
RedisURI redisUri = RedisURI.create("redis://authentication@localhost/2");
RedisClient client = RedisClient.create(rediUri);

// ···

client.shutdown();
```

4.3. Asynchronous API

This guide will give you an impression how and when to use the asynchronous API provided by lettuce 4.x.

4.3.1. Motivation

Asynchronous methodologies allow you to utilize better system resources, instead of wasting threads waiting for network or disk I/O. Threads can be fully utilized to perform other work instead. lettuce facilitates asynchronicity from building the client on top of netty that is a multithreaded, event-driven I/O framework. All communication is handled asynchronously. Once the foundation is able to processes commands concurrently, it is convenient to take advantage from the asynchronicity. It is way harder to turn a blocking and synchronous working software into a concurrently processing system.

Understanding Asynchronicity

Asynchronicity permits other processing to continue before the transmission has finished and the response of the transmission is processed. This means, in the context of lettuce and especially Redis, that multiple commands can be issued serially without the need of waiting to finish the preceding command. This mode of operation is also known as Pipelining. The following example should give you an impression of the mode of operation:

- Given client *A* and client *B*
- Client *A* triggers command `SET A=B`
- Client *B* triggers at the same time of Client *A* command `SET C=D`
- Redis receives command from Client *A*
- Redis receives command from Client *B*
- Redis processes `SET A=B` and responds `OK` to Client *A*
- Client *A* receives the response and stores the response in the response handle
- Redis processes `SET C=D` and responds `OK` to Client *B*
- Client *B* receives the response and stores the response in the response handle

Both clients from the example above can be either two threads or connections within an application or two physically separated clients.

Clients can operate concurrently to each other by either being separate processes, threads, event-loops, actors, fibers, etc. Redis processes incoming commands serially and operates mostly single-threaded. This means, commands are processed in the order they are received with some characteristic that we'll cover later.

Let's take the simplified example and enhance it by some program flow details:

- Given client *A*
- Client *A* triggers command `SET A=B`
- Client *A* uses the asynchronous API and can perform other processing
- Redis receives command from Client *A*
- Redis processes `SET A=B` and responds `OK` to Client *A*
- Client *A* receives the response and stores the response in the response handle
- Client *A* can access now the response to its command without waiting (non-blocking)

The Client *A* takes advantage from not waiting on the result of the command so it can process computational work or issue another Redis command. The client can work with the command result as soon as the response is available.

Impact of asynchronicity to the synchronous API

While this guide helps you to understand the asynchronous API it is worthwhile to learn the impact on the synchronous API. The general approach of the synchronous API is no different than the asynchronous API. In both cases, the same facilities are used to invoke and transport commands to the Redis server. The only difference is a blocking behavior of the caller that is using the synchronous API. Blocking happens on command level and affects only the command completion part, meaning multiple clients using the synchronous API can invoke commands on the same connection and at the same time without blocking each other. A call on the synchronous API is unblocked at the moment a command response was processed.

- Given client *A* and client *B*

- Client *A* triggers command `SET A=B` on the synchronous API and waits for the result
- Client *B* triggers at the same time of Client *A* command `SET C=D` on the synchronous API and waits for the result
- Redis receives command from Client *A*
- Redis receives command from Client *B*
- Redis processes `SET A=B` and responds `OK` to Client *A*
- Client *A* receives the response and unblocks the program flow of Client *A*
- Redis processes `SET C=D` and responds `OK` to Client *B*
- Client *B* receives the response and unblocks the program flow of Client *B*

However, there are some cases you should not share a connection among threads to avoid side-effects. The cases are:

- Disabling flush-after-command to improve performance
- The use of blocking operations like `BLPOP`. Blocking operations are queued on Redis until they can be executed. While one connection is blocked, other connections can issue commands to Redis. Once a command unblocks the blocking command (that said an `LPUSH` or `RPUSH` hits the list), the blocked connection is unblocked and can proceed after that.
- Transactions
- Using multiple databases

Result handles

Every command invocation on the asynchronous API creates a `RedisFuture<T>` that can be canceled, awaited and subscribed (listener). A `CompleteableFuture<T>` or `RedisFuture<T>` is a pointer to the result that is initially unknown since the computation of its value is yet incomplete. A `RedisFuture<T>` provides operations for synchronization and chaining.

Example 6. First steps with `CompletableFuture`

```
CompletableFuture<String> future = new CompletableFuture<>();

System.out.println("Current state: " + future.isDone());

future.complete("my value");

System.out.println("Current state: " + future.isDone());
System.out.println("Got value: " + future.get());
```

The example prints the following lines:

```
Current state: false
Current state: true
Got value: my value
```

Attaching a listener to a future allows chaining. Promises can be used synonymous to futures, but not every future is a promise. A promise guarantees a callback/notification and thus it has come to its name.

A simple listener that gets called once the future completes:

Example 7. Using listeners with `CompletableFuture`

```
final CompletableFuture<String> future = new CompletableFuture<>();

future.thenRun(new Runnable() {
    @Override
    public void run() {
        try {
            System.out.println("Got value: " + future.get());
        } catch (Exception e) {
            e.printStackTrace();
        }

    }
});

System.out.println("Current state: " + future.isDone());
future.complete("my value");
System.out.println("Current state: " + future.isDone());
```

The value processing moves from the caller into a listener that is then called by whoever completes the future. The example prints the following lines:

```
Current state: false
Got value: my value
Current state: true
```

The code from above requires exception handling since calls to the `get()` method can lead to exceptions. Exceptions raised during the computation of the `Future<T>` are transported within an `ExecutionException`. Another exception that may be thrown is the `InterruptedException`. This is because calls to `get()` are blocking calls and the blocked thread can be interrupted at any time. Just think about a system shutdown.

The `CompletionStage<T>` type allows since Java 8 a much more sophisticated handling of futures. A `CompletionStage<T>` can consume, transform and build a chain of value processing. The code from above can be rewritten in Java 8 in the following style:

Example 8. Using a `Consumer` future listener

```
CompletableFuture<String> future = new CompletableFuture<>();

future.thenAccept(new Consumer<String>() {
    @Override
    public void accept(String value) {
        System.out.println("Got value: " + value);
    }
});

System.out.println("Current state: " + future.isDone());
future.complete("my value");
System.out.println("Current state: " + future.isDone());
```

The example prints the following lines:

```
Current state: false
Got value: my value
Current state: true
```

You can find the full reference for the `CompletionStage<T>` type in the Java 8 API documentation.

4.3.2. Creating futures using lettuce

lettuce futures can be used for initial and chaining operations. When using lettuce futures, you will notice the non-blocking behavior. This is because all I/O and command processing are handled asynchronously using the netty EventLoop. The lettuce `RedisFuture<T>` extends a `CompletionStage<T>` so all methods of the base type are available.

lettuce exposes its futures on the Standalone, Sentinel, Publish/Subscribe and Cluster APIs.

Connecting to Redis is insanely simple:

```
RedisClient client = RedisClient.create("redis://localhost");
RedisAsyncCommands<String, String> commands = client.connect().async();
```

In the next step, obtaining a value from a key requires the `GET` operation:

```
RedisFuture<String> future = commands.get("key");
```

4.3.3. Consuming futures

The first thing you want to do when working with futures is to consume them. Consuming a futures means obtaining the value. Here is an example that blocks the calling thread and prints the value:

Example 9. `GET` a key

```
RedisFuture<String> future = commands.get("key");
String value = future.get();
System.out.println(value);
```

Invocations to the `get()` method (pull-style) block the calling thread at least until the value is computed but in the worst case indefinitely. Using timeouts is always a good idea to not exhaust your threads.

Example 10. Blocking synchronization

```
try {
    RedisFuture<String> future = commands.get("key");
    String value = future.get(1, TimeUnit.MINUTES);
    System.out.println(value);
} catch (Exception e) {
    e.printStackTrace();
}
```

The example will wait at most 1 minute for the future to complete. If the timeout exceeds, a `TimeoutException` is thrown to signal the timeout.

Futures can also be consumed in a push style, meaning when the `RedisFuture<T>` is completed, a follow-up action is triggered:

Example 11. Using a `Consumer` listener with `GET`

```
RedisFuture<String> future = commands.get("key");

future.thenAccept(new Consumer<String>() {
    @Override
    public void accept(String value) {
        System.out.println(value);
    }
});
```

Alternatively, written in Java 8 lambdas:

Example 12. Using a `Consumer` lambda with `GET`

```
RedisFuture<String> future = commands.get("key");

future.thenAccept(System.out::println);
```

lettuce futures are completed on the netty EventLoop. Consuming and chaining futures on the default thread is always a good idea except for one case: Blocking/long-running operations. As a rule of thumb, never block the event loop. If you need to chain futures using blocking calls, use the `thenAcceptAsync()`/`thenRunAsync()` methods to fork the processing to another thread. The `⋯async()` methods need a threading infrastructure for execution, by default the `ForkJoinPool.commonPool()` is used. The `ForkJoinPool` is statically constructed and does not grow with increasing load. Using default `Executor`s is almost always the better idea.

Example 13. Asynchronous listener notification

```
Executor sharedExecutor = ...
RedisFuture<String> future = commands.get("key");

future.thenAcceptAsync(new Consumer<String>() {
    @Override
    public void accept(String value) {
        System.out.println(value);
    }
}, sharedExecutor);
```

4.3.4. Synchronizing futures

A key point when using futures is the synchronization. Futures are usually used to:

1. Trigger multiple invocations without the urge to wait for the predecessors (Batching)
2. Invoking a command without awaiting the result at all (Fire&Forget)
3. Invoking a command and perform other computing in the meantime (Decoupling)
4. Adding concurrency to certain computational efforts (Concurrency)

There are several ways how to wait or get notified in case a future completes. Certain synchronization techniques apply to some motivations why you want to use futures.

Blocking synchronization

Blocking synchronization comes handy if you perform batching/add concurrency to certain parts of your system. An example to batching can be setting/retrieving multiple values and awaiting the results before a certain point within processing.

Example 14. Getting multiple keys asynchronously

```
List<RedisFuture<String>> futures = new ArrayList<RedisFuture<String>>();

for (int i = 0; i < 10; i++) {
    futures.add(commands.get("key-" + i, "value-" + i));
}

LettuceFutures.awaitAll(1, TimeUnit.MINUTES, futures.toArray(new RedisFuture
[futures.size()]));
```

The code from above does not wait until a certain command completes before it issues another one. The synchronization is done after all commands are issued. The example code can easily be turned into a Fire&Forget pattern by omitting the call to `LettuceFutures.awaitAll()`.

A single future execution can be also awaited, meaning an opt-in to wait for a certain time but without raising an exception:

Example 15. Using `RedisFuture.await` to wait for a result

```
RedisFuture<String> future = commands.get("key");

if(!future.await(1, TimeUnit.MINUTES)) {
    System.out.println("Could not complete within the timeout");
}
```

Calling `await()` is friendlier to call since it throws only an `InterruptedException` in case the blocked thread is interrupted. You are already familiar with the `get()` method for synchronization, so we will not bother you with this one.

At last, there is another way to synchronize futures in a blocking way. The major caveat is that you will become responsible to handle thread interruptions. If you do not handle that aspect, you will not be able to shut down your system properly if it is in a running state.

```
RedisFuture<String> future = commands.get("key");
while (!future.isDone()) {
    // do something ...
}
```

While the `isDone()` method does not aim primarily for synchronization use, it might come handy to perform other computational efforts while the command is executed.

Chaining synchronization

Futures can be synchronized/chained in a non-blocking style to improve thread utilization.

Chaining works very well in systems relying on event-driven characteristics. Future chaining builds up a chain of one or more futures that are executed serially, and every chain member handles a part in the computation. The `CompletionStage<T>` API offers various methods to chain and transform futures. A simple transformation of the value can be done using the `thenApply()` method:

Example 16. Future chaining

```
future.thenApply(new Function<String, Integer>() {
    @Override
    public Integer apply(String value) {
        return value.length();
    }
}).thenAccept(new Consumer<Integer>() {
    @Override
    public void accept(Integer integer) {
        System.out.println("Got value: " + integer);
    }
});
```

Alternatively, written in Java 8 lambdas:

Example 17. Future chaining with lambdas

```
future.thenApply(String::length)
    .thenAccept(integer -> System.out.println("Got value: " + integer));
```

The `thenApply()` method accepts a function that transforms the value into another one. The final `thenAccept()` method consumes the value for final processing.

You have already seen the `thenRun()` method from previous examples. The `thenRun()` method can be used to handle future completions in case the data is not crucial to your flow:

```
future.thenRun(new Runnable() {
    @Override
    public void run() {
        System.out.println("Finished the future.");
    }
});
```

Keep in mind to execute the `Runnable` on a custom `Executor` if you are doing blocking calls within the `Runnable`.

Another chaining method worth mentioning is the either-or chaining. A couple of `…Either()` methods are available on a `CompletionStage<T>`, see the Java 8 API docs for the full reference. The either-or pattern consumes the value from the first future that is completed. A good example might

be two services returning the same data, for instance, a Master-Slave scenario, but you want to return the data as fast as possible:

Example 18. Read from Master and Slave and continue with the first response

```
RedisStringAsyncCommands<String, String> master = masterClient.connect().async();
RedisStringAsyncCommands<String, String> slave = slaveClient.connect().async();

RedisFuture<String> future = master.get("key");
future.acceptEither(slave.get("key"), new Consumer<String>() {
    @Override
    public void accept(String value) {
      System.out.println("Got value: " + value);
    }
});
```

4.3.5. Error handling

Error handling is an indispensable component of every real world application and should to be considered from the beginning on. Futures provide some mechanisms to deal with errors.

In general, you want to react in the following ways:

- Return a default value instead
- Use a backup future
- Retry the future

`RedisFuture<T>`s transport exceptions if any occurred. Calls to the `get()` method throw the occurred exception wrapped within an `ExecutionException` (this is different to lettuce 3.x). You can find more details within the Javadoc on CompletionStage.

The following code falls back to a default value after it runs to an exception by using the `handle()` method:

Example 19. Future listener receiving result and error objects

```
future.handle(new BiFunction<String, Throwable, String>() {
    @Override
    public Integer apply(String value, Throwable throwable) {
        if(throwable != null) {
          return "default value";
        }
        return value;
    }
}).thenAccept(new Consumer<String>() {
    @Override
    public void accept(String value) {
        System.out.println("Got value: " + value);
    }
});
```

More sophisticated code could decide on behalf of the throwable type that value to return, as the shortcut example using the `exceptionally()` method:

Example 20. Future recovery with Exception handlers

```
future.exceptionally(new Function<Throwable, String>() {
    @Override
    public String apply(Throwable throwable) {
        if (throwable instanceof IllegalStateException) {
            return "default value";
        }

        return "other default value";
    }
});
```

Retrying futures and recovery using futures is not part of the Java 8 `CompleteableFuture<T>`. See the Reactive API for comfortable ways handling with exceptions.

4.3.6. Examples

Example 21. Basic operations

```
RedisAsyncCommands<String, String> async = client.connect().async();
RedisFuture<String> set = async.set("key", "value");
RedisFuture<String> get = async.get("key");

set.get() == "OK"
get.get() == "value"
```

Example 22. Waiting for a future with a timeout

```
RedisAsyncCommands<String, String> async = client.connect().async();
RedisFuture<String> set = async.set("key", "value");
RedisFuture<String> get = async.get("key");

set.await(1, SECONDS) == true
set.get() == "OK"
get.get(1, TimeUnit.MINUTES) == "value"
```

Example 23. Using a listener with `RedisFuture`

```
RedisStringAsyncCommands<String, String> async = client.connect().async();
RedisFuture<String> set = async.set("key", "value");

Runnable listener = new Runnable() {
    @Override
    public void run() {
            ...;
    }
};

set.thenRun(listener);
```

4.4. Reactive API

This guide helps you to understand the Reactive Stream pattern and aims to give you a general understanding of how to build reactive applications.

4.4.1. Motivation

Asynchronous and reactive methodologies allow you to utilize better system resources, instead of wasting threads waiting for network or disk I/O. Threads can be fully utilized to perform other work instead.

A broad range of technologies exists to facilitate this style of programming, ranging from the very limited and less usable `java.util.concurrent.Future` to complete libraries and runtimes like Akka. Project Reactor, has a very rich set of operators to compose asynchronous workflows, it has no further dependencies to other frameworks and supports the very mature Reactive Streams model.

4.4.2. Understanding Reactive Streams

Reactive Streams is an initiative to provide a standard for asynchronous stream processing with non-blocking back pressure. This encompasses efforts aimed at runtime environments (JVM and JavaScript) as well as network protocols.

The scope of Reactive Streams is to find a minimal set of interfaces, methods, and protocols that will describe the necessary operations and entities to achieve the goal—asynchronous streams of data with non-blocking back pressure.

It is an interoperability standard between multiple reactive composition libraries that allow interaction without the need of bridging between libraries in application code.

The integration of Reactive Streams is usually accompanied with the use of a composition library that hides the complexity of bare `Publisher<T>` and `Subscriber<T>` types behind an easy-to-use API. Lettuce uses Project Reactor that exposes its publishers as `Mono` and `Flux`.

For more information about Reactive Streams see http://reactive-streams.org.

4.4.3. Understanding Publishers

Asynchronous processing decouples I/O or computation from the thread that invoked the operation. A handle to the result is given back, usually a `java.util.concurrent.Future` or similar, that returns either a single object, a collection or an exception. Retrieving a result, that was fetched asynchronously is usually not the end of processing one flow. Once data is obtained, further requests can be issued, either always or conditionally. With Java 8 or the Promise pattern, linear chaining of futures can be set up so that subsequent asynchronous requests are issued. Once conditional processing is needed, the asynchronous flow has to be interrupted and synchronized. While this approach is possible, it does not fully utilize the advantage of asynchronous processing.

In contrast to the preceding examples, `Publisher<T>` objects answer the multiplicity and asynchronous questions in a different fashion: By inverting the `Pull` pattern into a `Push` pattern.

A Publisher is the asynchronous/push “dual” to the synchronous/pull Iterable

event	Iterable (pull)	Publisher (push)
retrieve data	T next()	onNext(T)
discover error	throws Exception	onError(Exception)
complete	!hasNext()	onCompleted()

An `Publisher<T>` supports emission sequences of values or even infinite streams, not just the emission of single scalar values (as Futures do). You will very much appreciate this fact once you start to work on streams instead of single values. Project Reactor uses two types in its vocabulary: `Mono` and `Flux` that are both publishers.

A `Mono` can emit `0` to `1` events while a `Flux` can emit `0` to `N` events.

A `Publisher<T>` is not biased toward some particular source of concurrency or asynchronicity and how the underlying code is executed - synchronous or asynchronous, running within a `ThreadPool`. As a consumer of a `Publisher<T>`, you leave the actual implementation to the supplier, who can change it later on without you having to adapt your code.

The last key point of a `Publisher<T>` is that the underlying processing is not started at the time the `Publisher<T>` is obtained, rather its started at the moment an observer subscribes or signals demand to the `Publisher<T>`. This is a crucial difference to a `java.util.concurrent.Future`, which is started somewhere at the time it is created/obtained. So if no observer ever subscribes to the `Publisher<T>`, nothing ever will happen.

4.4.4. A word on the lettuce Reactive API

All commands return a `Flux<T>`, `Mono<T>` or `Mono<Void>` to which a `Subscriber` can subscribe to. That subscriber reacts to whatever item or sequence of items the `Publisher<T>` emits. This pattern facilitates concurrent operations because it does not need to block while waiting for the `Publisher<T>` to emit objects. Instead, it creates a sentry in the form of a `Subscriber` that stands ready to react appropriately at whatever future time the `Publisher<T>` does so.

4.4.5. Consuming `Publisher<T>`

The first thing you want to do when working with publishers is to consume them. Consuming a publisher means subscribing to it. Here is an example that subscribes and prints out all the items emitted:

```
Flux.just("Ben", "Michael", "Mark").subscribe(new Subscriber<String>() {
    public void onSubscribe(Subscription s) {
        s.request(3);
    }

    public void onNext(String s) {
        System.out.println("Hello " + s + "!");
    }

    public void onError(Throwable t) {

    }

    public void onComplete() {
        System.out.println("Completed");
    }
});
```

The example prints the following lines:

```
Hello Ben
Hello Michael
Hello Mark
Completed
```

You can see that the Subscriber (or Observer) gets notified of every event and also receives the completed event. A `Publisher<T>` emits items until either an exception is raised or the `Publisher<T>` finishes the emission calling `onCompleted`. No further elements are emitted after that time.

A call to the `subscribe` registers a `Subscription` that allows to cancel and, therefore, do not receive further events. Publishers can interoperate with the un-subscription and free resources once a subscriber unsubscribed from the `Publisher<T>`.

Implementing a `Subscriber<T>` requires implementing numerous methods, so lets rewrite the code to a simpler form:

```
Flux.just("Ben", "Michael", "Mark").doOnNext(new Consumer<String>() {
    public void accept(String s) {
        System.out.println("Hello " + s + "!");
    }
}).doOnComplete(new Runnable() {
    public void run() {
        System.out.println("Completed");
    }
}).subscribe();
```

alternatively, even simpler by using Java 8 Lambdas:

```
Flux.just("Ben", "Michael", "Mark")
        .doOnNext(s -> System.out.println("Hello " + s + "!"))
        .doOnComplete(() -> System.out.println("Completed"))
        .subscribe();
```

You can control the elements that are processed by your `Subscriber` using operators. The `take()` operator limits the number of emitted items if you are interested in the first `N` elements only.

```
Flux.just("Ben", "Michael", "Mark") //
        .doOnNext(s -> System.out.println("Hello " + s + "!"))
        .doOnComplete(() -> System.out.println("Completed"))
        .take(2)
        .subscribe();
```

The example prints the following lines:

```
Hello Ben
Hello Michael
Completed
```

Note that the `take` operator implicitly cancels its subscription from the `Publisher<T>` once the expected count of elements was emitted.

A subscription to a `Publisher<T>` can be done either by another `Flux` or a `Subscriber`. Unless you are implementing a custom `Publisher`, always use `Subscriber`. The used subscriber `Consumer` from the example above does not handle `Exception`s so once an `Exception` is thrown you will see a stack trace like this:

```
Exception in thread "main" reactor.core.Exceptions$BubblingException:
java.lang.RuntimeException: Example exception
    at reactor.core.Exceptions.bubble(Exceptions.java:96)
    at reactor.core.publisher.Operators.onErrorDropped(Operators.java:296)
    at reactor.core.publisher.LambdaSubscriber.onError(LambdaSubscriber.java:117)
    ...
Caused by: java.lang.RuntimeException: Example exception
    at demos.lambda$example3Lambda$4(demos.java:87)
    at
reactor.core.publisher.FluxPeekFuseable$PeekFuseableSubscriber.onNext(FluxPeekFuseable
.java:157)
    ... 23 more
```

It is always recommended to implement an error handler right from the beginning. At a certain point, things can and will go wrong.

A fully implemented subscriber declares the `onCompleted` and `onError` methods allowing you to react to these events:

```
Flux.just("Ben", "Michael", "Mark").subscribe(new Subscriber<String>() {
    public void onSubscribe(Subscription s) {
        s.request(3);
    }

    public void onNext(String s) {
        System.out.println("Hello " + s + "!");
    }

    public void onError(Throwable t) {
        System.out.println("onError: " + e);
    }

    public void onComplete() {
        System.out.println("Completed");
    }
});
```

4.4.6. From push to pull

The examples from above illustrated how publishers can be set up in a not-opinionated style about blocking or non-blocking execution. A `Flux<T>` can be converted explicitly into an `Iterable<T>` or synchronized with `block()`. Avoid calling `block()` in your code as you start expressing the nature of execution inside your code. Calling `block()` removes all non-blocking advantages of the reactive chain to your application.

```
String last = Flux.just("Ben", "Michael", "Mark").last().block();
System.out.println(last);
```

The example prints the following line:

```
Mark
```

A blocking call can be used to synchronize the publisher chain and find back a way into the plain and well-known `Pull` pattern.

```
List<String> list = Flux.just("Ben", "Michael", "Mark").collectList().block();
System.out.println(list);
```

The `toList` operator collects all emitted elements and passes the list through the `BlockingPublisher<T>`.

The example prints the following line:

```
[Ben, Michael, Mark]
```

4.4.7. Creating `Flux` and `Mono` using lettuce

There are many ways to establish publishers. You have already seen `just()`, `take()` and `collectList()`. Refer to the Project Reactor documentation for many more methods that you can use to create `Flux` and `Mono`.

lettuce publishers can be used for initial and chaining operations. When using lettuce publishers, you will notice the non-blocking behavior. This is because all I/O and command processing are handled asynchronously using the netty EventLoop.

Connecting to Redis is insanely simple:

```
RedisClient client = RedisClient.create("redis://localhost");
RedisStringReactiveCommands<String, String> commands = client.connect().reactive();
```

In the next step, obtaining a value from a key requires the `GET` operation:

```
commands.get("key").subscribe(new Consumer<String>() {

    public void accept(String value) {
        System.out.println(value);
    }
});
```

Alternatively, written in Java 8 lambdas:

```
commands
   .get("key")
   .subscribe(value -> System.out.println(value));
```

The execution is handled asynchronously, and the invoking Thread can be used to processed in processing while the operation is completed on the Netty EventLoop threads. Due to its decoupled nature, the calling method can be left before the execution of the `Publisher<T>` is finished.

lettuce publishers can be used within the context of chaining to load multiple keys asynchronously:

```
Flux.just("Ben", "Michael", "Mark").
        flatMap(key -> commands.get(key)).
        subscribe(value -> System.out.println("Got value: " + value));
```

4.4.8. Hot and Cold Publishers

There is a distinction between Publishers that was not covered yet:

- A cold Publishers waits for a subscription until it emits values and does this freshly for every subscriber.
- A hot Publishers begins emitting values upfront and presents them to every subscriber subsequently.

All Publishers returned from the Redis Standalone, Redis Cluster, and Redis Sentinel API are cold, meaning that no I/O happens until they are subscribed to. As such a subscriber is guaranteed to see the whole sequence from the beginning. So just creating a Publisher will not cause any network I/O thus creating and discarding Publishers is cheap. Publishers created for a Publish/Subscribe emit `PatternMessage`s and `ChannelMessage`s once they are subscribed to. Publishers guarantee however to emit all items from the beginning until their end. While this is true for Publish/Subscribe publishers, the nature of subscribing to a Channel/Pattern allows missed messages due to its subscription nature and less to the Hot/Cold distinction of publishers.

4.4.9. Transforming publishers

Publishers can transform the emitted values in various ways. One of the most basic transformations is `flatMap()` which you have seen from the examples above that converts the incoming value into a different one. Another one is `map()`. The difference between `map()` and `flatMap()` is that `flatMap()` allows you to do those transformations with `Publisher<T>` calls.

```
Flux.just("Ben", "Michael", "Mark")
        .flatMap(commands::get)
        .flatMap(value -> commands.rpush("result", value))
        .subscribe();
```

The first `flatMap()` function is used to retrieve a value and the second `flatMap()` function appends the value to a Redis list named `result`. The `flatMap()` function returns a Publisher whereas the normal map just returns `<T>`. You will use `flatMap()` a lot when dealing with flows like this, you'll become good friends.

An aggregation of values can be achieved using the `reduce()` transformation. It applies a function to each value emitted by a `Publisher<T>`, sequentially and emits each successive value. We can use it to aggregate values, to count the number of elements in multiple Redis sets:

```
Flux.just("Ben", "Michael", "Mark")
        .flatMap(commands::scard)
        .reduce((sum, current) -> sum + current)
        .subscribe(result -> System.out.println("Number of elements in sets: " +
result));
```

The aggregation function of `reduce()` is applied on each emitted value, so three times in the example above. If you want to get the last value, which denotes the final result containing the

number of elements in all Redis sets, apply the `last()` transformation:

```
Flux.just("Ben", "Michael", "Mark")
        .flatMap(commands::scard)
        .reduce((sum, current) -> sum + current)
        .last()
        .subscribe(result -> System.out.println("Number of elements in sets: " +
result));
```

Now let's take a look at grouping emitted items. The following example emits three items and groups them by the beginning character.

```
Flux.just("Ben", "Michael", "Mark")
    .groupBy(key -> key.substring(0, 1))
    .subscribe(
        groupedFlux -> {
            groupedFlux.collectList().subscribe(list -> {
                System.out.println("First character: " + groupedFlux.key() + ",
elements: " + list);
            });
        }
);
```

The example prints the following lines:

```
First character: B, elements: [Ben]
First character: M, elements: [Michael, Mark]
```

4.4.10. Absent values

The presence and absence of values is an essential part of reactive programming. Traditional approaches consider `null` as an absence of a particular value. With Java 8, `Optional<T>` was introduced to encapsulate nullability. Reactive Streams prohibits the use of `null` values.

In the scope of Redis, an absent value is an empty list, a non-existent key or any other empty data structure. Reactive programming discourages the use of `null` as value. The reactive answer to absent values is just not emitting any value that is possible due the `0` to `N` nature of `Publisher<T>`.

Suppose we have the keys `Ben` and `Michael` set each to the value `value`. We query those and another, absent key with the following code:

```
Flux.just("Ben", "Michael", "Mark")
        .flatMap(commands::get)
        .doOnNext(value -> System.out.println(value))
        .subscribe();
```

The example prints the following lines:

```
value
value
```

The output is just two values. The `GET` to the absent key `Mark` does not emit a value.

The reactive API provides operators to work with empty results when you require a value. You can use one of the following operators:

- `defaultIfEmpty`: Emit a default value if the `Publisher<T>` did not emit any value at all
- `switchIfEmpty`: Switch to a fallback `Publisher<T>` to emit values
- `Flux.hasElements`/`Flux.hasElement`: Emit a `Mono<Boolean>` that contains a flag whether the original `Publisher<T>` is empty
- `next`/`last`/`elementAt`: Positional operators to retrieve the first/last/Nth element or emit a default value

4.4.11. Filtering items

The values emitted by a `Publisher<T>` can be filtered in case you need only specific results. Filtering does not change the emitted values itself. Filters affect how many items and at which point (and if at all) they are emitted.

```
Flux.just("Ben", "Michael", "Mark")
        .filter(s -> s.startsWith("M"))
        .flatMap(commands::get)
        .subscribe(value -> System.out.println("Got value: " + value));
```

The code will fetch only the keys `Michael` and `Mark` but not `Ben`. The filter criteria are whether the `key` starts with a `M`.

You already met the `last()` filter to retrieve the last value:

```
Flux.just("Ben", "Michael", "Mark")
        .last()
        .subscribe(value -> System.out.println("Got value: " + value));
```

the extended variant of `last()` allows you to take the last `N` values:

```
Flux.just("Ben", "Michael", "Mark")
        .takeLast(3)
        .subscribe(value -> System.out.println("Got value: " + value));
```

The example from above takes the last `2` values.

The opposite to `next()` is the `first()` filter that is used to retrieve the next value:

```
Flux.just("Ben", "Michael", "Mark")
        .next()
        .subscribe(value -> System.out.println("Got value: " + value));
```

4.4.12. Error handling

Error handling is an indispensable component of every real world application and should to be considered from the beginning on. Project Reactor provides several mechanisms to deal with errors.

In general, you want to react in the following ways:

- Return a default value instead
- Use a backup publisher
- Retry the Publisher (immediately or with delay)

The following code falls back to a default value after it throws an exception at the first emitted item:

```
Flux.just("Ben", "Michael", "Mark")
        .doOnNext(value -> {
            throw new IllegalStateException("Takes way too long");
        })
        .onErrorReturn("Default value")
        .subscribe();
```

You can use a backup `Publisher<T>` which will be called if the first one fails.

```
Flux.just("Ben", "Michael", "Mark")
        .doOnNext(value -> {
            throw new IllegalStateException("Takes way too long");
        })
        .switchOnError(commands.get("Default Key"))
        .subscribe();
```

It is possible to retry the publisher by re-subscribing. Re-subscribing can be done as soon as possible, or with a wait interval, which is preferred when external resources are involved.

```
Flux.just("Ben", "Michael", "Mark")
        .flatMap(commands::get)
        .retry()
        .subscribe();
```

Use the following code if you want to retry with backoff:

```
Flux.just("Ben", "Michael", "Mark")
        .doOnNext(v -> {
            if (new Random().nextInt(10) + 1 == 5) {
                throw new RuntimeException("Boo!");
            }
        })
        .doOnSubscribe(subscription ->
        {
            System.out.println(subscription);
        })
        .retryWhen(throwableFlux -> Flux.range(1, 5)
                .flatMap(i -> {
                    System.out.println(i);
                    return Flux.just(i)
                            .delay(Duration.of(i, ChronoUnit.SECONDS));
                }))
        .blockLast();
```

The attempts get passed into the `retryWhen()` method delayed with the number of seconds to wait. The delay method is used to complete once its timer is done.

4.4.13. Schedulers and threads

Schedulers in Project Reactor are used to instruct multi-threading. Some operators have variants that take a Scheduler as a parameter. These instruct the operator to do some or all of its work on a particular Scheduler.

Project Reactor ships with a set of preconfigured Schedulers, which are all accessible through the `Schedulers` class:

- Schedulers.parallel(): Executes the computational work such as event-loops and callback processing.
- Schedulers.immediate(): Executes the work immediately in the current thread
- Schedulers.elastic(): Executes the I/O-bound work such as asynchronous performance of blocking I/O, this scheduler is backed by a thread-pool that will grow as needed
- Schedulers.newSingle(): Executes the work on a new thread
- Schedulers.fromExecutor(): Create a scheduler from a `java.util.concurrent.Executor`
- Schedulers.timer(): Create or reuse a hash-wheel based TimedScheduler with a resolution of 50ms.

Do not use the computational scheduler for I/O.

Publishers can be executed by a scheduler in the following different ways:

- Using an operator that makes use of a scheduler

- Explicitly by passing the Scheduler to such an operator
- By using `subscribeOn(Scheduler)`
- By using `publishOn(Scheduler)`

Operators like `buffer`, `replay`, `skip`, `delay`, `parallel`, and so forth use a Scheduler by default if not instructed otherwise.

All of the listed operators allow you to pass in a custom scheduler if needed. Sticking most of the time with the defaults is a good idea.

If you want the subscribe chain to be executed on a specific scheduler, you use the `subscribeOn()` operator. The code is executed on the main thread without a scheduler set:

```
Flux.just("Ben", "Michael", "Mark").flatMap(key -> {
            System.out.println("Map 1: " + key + " (" + Thread.currentThread().
getName() + ")");
            return Flux.just(key);
        }
).flatMap(value -> {
            System.out.println("Map 2: " + value + " (" + Thread.currentThread()
.getName() + ")");
            return Flux.just(value);
        }
).subscribe();
```

The example prints the following lines:

```
Map 1: Ben (main)
Map 2: Ben (main)
Map 1: Michael (main)
Map 2: Michael (main)
Map 1: Mark (main)
Map 2: Mark (main)
```

This example shows the `subscribeOn()` method added to the flow (it does not matter where you add it):

```
Flux.just("Ben", "Michael", "Mark").flatMap(key -> {
            System.out.println("Map 1: " + key + " (" + Thread.currentThread().
getName() + ")");
            return Flux.just(key);
        }
).flatMap(value -> {
            System.out.println("Map 2: " + value + " (" + Thread.currentThread()
.getName() + ")");
            return Flux.just(value);
        }
).subscribeOn(Schedulers.parallel()).subscribe();
```

The output of the example shows the effect of `subscribeOn()`. You can see that the Publisher is executed on the same thread, but on the computation thread pool:

```
Map 1: Ben (parallel-1)
Map 2: Ben (parallel-1)
Map 1: Michael (parallel-1)
Map 2: Michael (parallel-1)
Map 1: Mark (parallel-1)
Map 2: Mark (parallel-1)
```

If you apply the same code to lettuce, you will notice a difference in the threads on which the second `flatMap()` is executed:

```
Flux.just("Ben", "Michael", "Mark").flatMap(key -> {
    System.out.println("Map 1: " + key + " (" + Thread.currentThread().getName() + ")
");
    return commands.set(key, key);
}).flatMap(value -> {
    System.out.println("Map 2: " + value + " (" + Thread.currentThread().getName() +
")");
    return Flux.just(value);
}).subscribeOn(Schedulers.parallel()).subscribe();
```

The example prints the following lines:

```
Map 1: Ben (parallel-1)
Map 1: Michael (parallel-1)
Map 1: Mark (parallel-1)
Map 2: OK (lettuce-nioEventLoop-3-1)
Map 2: OK (lettuce-nioEventLoop-3-1)
Map 2: OK (lettuce-nioEventLoop-3-1)
```

Two things differ from the standalone examples:

1. The values are set rather concurrently than sequentially
2. The second `flatMap()` transformation prints the netty EventLoop thread name

This is because lettuce publishers are executed and completed on the netty EventLoop threads by default.

`publishOn` instructs an Publisher to call its observer's `onNext`, `onError`, and `onCompleted` methods on a particular Scheduler. Here, the order matters:

```
Flux.just("Ben", "Michael", "Mark").flatMap(key -> {
    System.out.println("Map 1: " + key + " (" + Thread.currentThread().getName() + ")
");
    return commands.set(key, key);
}).publishOn(Schedulers.parallel()).flatMap(value -> {
    System.out.println("Map 2: " + value + " (" + Thread.currentThread().getName() +
")");
    return Flux.just(value);
}).subscribe();
```

Everything before the `publishOn()` call is executed in main, everything below in the scheduler:

```
Map 1: Ben (main)
Map 1: Michael (main)
Map 1: Mark (main)
Map 2: OK (parallel-1)
Map 2: OK (parallel-1)
Map 2: OK (parallel-1)
```

Schedulers allow direct scheduling of operations. Refer to the Project Reactor documentation for further information.

4.4.14. Redis Transactions

lettuce provides a convenient way to use Redis Transactions in a reactive way. Commands that should be executed within a transaction can be executed after the `MULTI` command was executed. Functional chaining allows to execute commands within a closure, and each command receives its appropriate response. A cumulative response is also returned with `TransactionResult` in response to `EXEC`.

See Transactions for further details.

Other examples

Blocking example

```
RedisStringReactiveCommands<String, String> reactive = client.connect().reactive();
Mono<String> set = reactive.set("key", "value");
set.block();
```

Non-blocking example

```
RedisStringReactiveCommands<String, String> reactive = client.connect().reactive();
Mono<String> set = reactive.set("key", "value");
set.subscribe();
```

Functional chaining

```
RedisStringReactiveCommands<String, String> reactive = client.connect().reactive();
Flux.just("Ben", "Michael", "Mark")
        .flatMap(key -> commands.sadd("seen", key))
        .flatMap(value -> commands.randomkey())
        .flatMap(commands::type)
        .doOnNext(System.out::println).subscribe();
```

Redis Transaction

```
RedisReactiveCommands<String, String> reactive = client.connect().reactive();

reactive.multi().doOnSuccess(s -> {
    reactive.set("key", "1").doOnNext(s1 -> System.out.println(s1)).subscribe();
    reactive.incr("key").doOnNext(s1 -> System.out.println(s1)).subscribe();
}).flatMap(s -> reactive.exec())
        .doOnNext(transactionResults ->
System.out.println(transactionResults.wasRolledBack()))
        .subscribe();
```

4.5. Publish/Subscribe

lettuce provides support for Publish/Subscribe on Redis Standalone and Redis Cluster connections. The connection is notified on message/subscribed/unsubscribed events after subscribing to channels or patterns. Synchronous, asynchronous and reactive API's are provided to interact with Redis Publish/Subscribe features.

4.5.1. Subscribing

A connection can notify multiple listeners that implement `RedisPubSubListener` (lettuce provides a `RedisPubSubAdapter` for convenience). All listener registrations are kept within the `StatefulRedisPubSubConnection`/`StatefulRedisClusterConnection`.

Example 24. Synchronous subscription

```
StatefulRedisPubSubConnection<String, String> connection = client.connectPubSub()
connection.addListener(new RedisPubSubListener<String, String>() { ... })

RedisPubSubCommands<String, String> sync = connection.sync();
sync.subscribe("channel");

// application flow continues
```

Example 25. Asynchronous subscription

```
StatefulRedisPubSubConnection<String, String> connection = client.connectPubSub()
connection.addListener(new RedisPubSubListener<String, String>() { ... })

RedisPubSubAsyncCommands<String, String> async = connection.async();
RedisFuture<Void> future = async.subscribe("channel");

// application flow continues
```

4.5.2. Reactive API

The reactive API provides hot `Observable`s to listen on `ChannelMessage`s and `PatternMessage`s. The `Observable`s receive all inbound messages. You can do filtering using the observable chain if you need to filter out the interesting ones, The `Observable` stops triggering events when the subscriber unsubscribes from it.

Example 26. Reactive subscription

```
StatefulRedisPubSubConnection<String, String> connection = client.connectPubSub()

RedisPubSubReactiveCommands<String, String> reactive = connection.reactive();
reactive.subscribe("channel").subscribe();

reactive.observeChannels().doOnNext(patternMessage -> {...}).subscribe()

// application flow continues
```

Example 27. Redis Cluster Publish/Subscribe

```
StatefulRedisClusterConnection<String, String> connection = clusterClient
.connectPubSub()
connection.addListener(new RedisPubSubListener<String, String>() { ... })

RedisPubSubCommands<String, String> sync = connection.sync();
sync.subscribe("channel");
```

4.6. Transactions/Multi

Transactions allow the execution of a group of commands in a single step. Transactions can be controlled using `WATCH`, `UNWATCH`, `EXEC`, `MULTI` and `DISCARD` commands. Synchronous, asynchronous, reactive and cluster API's allow the use of transactions.

Redis responds to commands invoked during a transaction with a with `QUEUED` response. The response related to the execution of the command is received at the moment the `EXEC` command is processed, and the transaction is executed. The particular APIs behave in different ways:

- Synchronous: Invocations to the commands return `null` while they are invoked within a transaction. The `MULTI` command carries the response of the particular commands.
- Asynchronous: The futures receive their response at the moment the `EXEC` command is processed. This happens while the `EXEC` response is received.
- Reactive: An `Obervable<T>` triggers `onNext`/`onCompleted` at the moment the `EXEC` command is processed. This happens while the `EXEC` response is received.

As soon as you're within a transaction, you won't receive any responses on triggering the commands

```
redis.multi() == "OK"
redis.set(key, value) == null
redis.exec() == list("OK")
```

You'll receive the transactional response when calling `exec()` on the end of your transaction.

```
redis.multi() == "OK"
redis.set(key1, value) == null
redis.set(key2, value) == null
redis.exec() == list("OK", "OK")
```

4.6.1. Transactions using the asynchronous API

Asynchronous use of Redis transactions is very similar to non-transactional use. The asynchronous API returns `RedisFuture` instances that eventually complete and they are handles to a future result.

Regular commands complete as soon as Redis sends a response. Transactional commands complete as soon as the `EXEC` result is received.

Each command is completed individually with its own result so users of `RedisFuture` will see no difference between transactional and non-transactional `RedisFuture` completion. That said, transactional command results are available twice: Once via `RedisFuture` of the command and once through `List<Object>` (`TransactionResult` since Lettuce 5) of the `EXEC` command future.

```
RedisAsyncCommands<String, String> async = client.connect().async();

RedisFuture<String> multi = async.multi();

RedisFuture<String> set = async.set("key", "value");

RedisFuture<List<Object>> exec = async.exec();

List<Object> objects = exec.get();
String setResult = set.get();

objects.get(0) == setResult
```

4.6.2. Transactions using the reactive API

The reactive API can be used to execute multiple commands in a single step. The nature of the reactive API encourages nesting of commands. It is essential to understand the time at which an `Observable<T>` emits a value when working with transactions. Redis responds with `QUEUED` to commands invoked during a transaction. The response related to the execution of the command is received at the moment the `EXEC` command is processed, and the transaction is executed. Subsequent calls in the processing chain are executed after the transactional end. The following code starts a transaction, executes two commands within the transaction and finally executes the transaction.

```
RedisReactiveCommands<String, String> reactive = client.connect().reactive();
reactive.multi().subscribe(multiResponse -> {
    reactive.set("key", "1").subscribe();
    reactive.incr("key").subscribe();
    reactive.exec().subscribe();
});
```

4.6.3. Transactions on clustered connections

Clustered connections perform a routing by default. This means, that you can't be really sure, on which host your command is executed. So if you are working in a clustered environment, use rather a regular connection to your node, since then you'll bound to that node knowing which hash slots are handled by it.

4.6.4. Examples

Multi with executing multiple commands

```
redis.multi();

redis.set("one", "1");
redis.set("two", "2");
redis.mget("one", "two");
redis.llen(key);

redis.exec(); // result: list("OK", "OK", list("1", "2"), 0L)
```

Mult executing multiple asynchronous commands

```
redis.multi();

RedisFuture<String> set1 = redis.set("one", "1");
RedisFuture<String> set2 = redis.set("two", "2");
RedisFuture<String> mget = redis.mget("one", "two");
RedisFuture<Long> llen = mgetredis.llen(key);


set1.thenAccept(value -> …); // OK
set2.thenAccept(value -> …); // OK

RedisFuture<List<Object>> exec = redis.exec(); // result: list("OK", "OK", list("1",
"2"), 0L)

mget.get(); // list("1", "2")
llen.thenAccept(value -> …); // 0L
```

Using WATCH

```
redis.watch(key);

RedisConnection<String, String> redis2 = client.connect();
redis2.set(key, value + "X");
redis2.close();

redis.multi();
redis.append(key, "foo");
redis.exec()); // result is a empty list because of the changed key
```

Chapter 5. High-Availability and Sharding

5.1. Master/Slave

Redis can increase availability and read throughput by using replication. lettuce provides dedicated Master/Slave support since 4.2 for topologies and ReadFrom-Settings.

Redis Master/Slave can be run standalone or together with Redis Sentinel, which provides automated failover and master promotion. Failover and master promotion is supported in lettuce already since version 3.1 for master connections.

Connections can be obtained from the `MasterSlave` connection provider by supplying the client, Codec, and one or multiple RedisURIs.

5.1.1. Redis Sentinel

Master/Slave using Redis Sentinel uses Redis Sentinel as registry and notification source for topology events. Details about the master and its slaves are obtained from Redis Sentinel. lettuce subscribes to Redis Sentinel events for notifications to all supplied Sentinels.

5.1.2. Standalone Master/Slave

Running a Standalone Master/Slave setup required one seed address to establish a Redis connection. Providing one `RedisURI` will discover other nodes which belong to the Master/Slave setup and use the discovered addresses for connections. The initial URI can point either to a master or a slave node.

5.1.3. Static Master/Slave with predefined node addresses

In some cases, topology discovery shouldn't be enabled, or the discovered Redis addresses are not suited for connections. AWS ElastiCache falls into this category. lettuce allows to specify one or more Redis addresses as `List` and predefine the node topology. Master/Slave URIs will be treated in this case as static topology, and no additional hosts are discovered in such case. Redis Standalone Master/Slave will discover the roles of the supplied `RedisURI`s and issue commands to the appropriate node.

5.1.4. Topology discovery

Master-Slave topologies are either static or semi-static. Redis Standalone instances with attached slaves provide no failover/HA mechanism. Redis Sentinel managed instances are controlled by Redis Sentinel and allow failover (which include master promotion). The `MasterSlave` API supports both mechanisms. The topology is provided by a `TopologyProvider`:

- `MasterSlaveTopologyProvider`: Dynamic topology lookup using the `INFO REPLICATION` output. Slaves are listed as slaveN=... entries. The initial connection can either point to a master or a slave, and the topology provider will discover nodes. The connection needs to be re-established outside of lettuce in a case of a Master/Slave failover or topology changes.

- `StaticMasterSlaveTopologyProvider`: Topology is defined by the list of URIs and the ROLE output. MasterSlave uses only the supplied nodes and won't discover additional nodes in the setup. The connection needs to be re-established outside of lettuce in case of a Master/Slave failover or topology changes.
- `SentinelTopologyProvider`: Dynamic topology lookup using the Redis Sentinel API. In particular, `SENTINEL MASTER` and `SENTINEL SLAVES` output. Master/Slave failover is handled by lettuce.

5.1.5. Topology Updates

- Standalone Master/Slave: Performs a one-time topology lookup which remains static afterward
- Redis Sentinel: Subscribes to all Sentinels and listens for Pub/Sub messages to trigger topology refreshing

Examples

Example 28. Redis Standalone Master/Slave

```
RedisClient redisClient = RedisClient.create();

StatefulRedisMasterSlaveConnection<String, String> connection = MasterSlave
.connect(redisClient, new Utf8StringCodec(),
        RedisURI.create("redis://localhost"));
connection.setReadFrom(ReadFrom.MASTER_PREFERRED);

System.out.println("Connected to Redis");

connection.close();
redisClient.shutdown();
```

Example 29. Redis Sentinel

```
RedisClient redisClient = RedisClient.create();

StatefulRedisMasterSlaveConnection<String, String> connection = MasterSlave
.connect(redisClient, new Utf8StringCodec(),
        RedisURI.create("redis-
sentinel://localhost:26379,localhost:26380/0#mymaster"));
connection.setReadFrom(ReadFrom.MASTER_PREFERRED);

System.out.println("Connected to Redis");

connection.close();
redisClient.shutdown();
```

Example 30. AWS ElastiCache Cluster

```
RedisClient redisClient = RedisClient.create();

List<RedisURI> nodes = Arrays.asList(RedisURI.create("redis://host1"),
        RedisURI.create("redis://host2"),
        RedisURI.create("redis://host3"));

StatefulRedisMasterSlaveConnection<String, String> connection = MasterSlave
        .connect(redisClient, new Utf8StringCodec(), nodes);
connection.setReadFrom(ReadFrom.MASTER_PREFERRED);

System.out.println("Connected to Redis");

connection.close();
redisClient.shutdown();
```

5.2. Redis Sentinel

When using lettuce, you can interact with Redis Sentinel and Redis Sentinel-managed nodes in multiple ways:

1. Direct connection to Redis Sentinel, for issuing Redis Sentinel commands
2. Using Redis Sentinel to connect to a master
3. Using Redis Sentinel to connect to masters and slaves through the Master/Slave.

In both cases, you need to supply a `RedisURI` since the Redis Sentinel integration supports multiple Sentinel hosts to provide high availability.

Please note: Redis Sentinel (lettuce 3.x) integration provides only asynchronous connections and no connection pooling.

5.2.1. Direct connection Redis Sentinel nodes

Lettuce exposes an API to interact with Redis Sentinel nodes directly. This is useful for performing administrative tasks using lettuce. You can monitor new masters, query master addresses, slaves and much more. A connection to a Redis Sentinel node is established by `RedisClient.connectSentinel()`. Use a Publish/Subscribe connection to subscribe to Sentinel events.

5.2.2. Redis discovery using Redis Sentinel

One or more Redis Sentinels can monitor Redis instances . These Redis instances are usually operated together with a slave of the Redis instance. Once the master goes down, the slave is promoted to a master. Once a master instance is not reachable anymore, the failover process is started by the Redis Sentinels. Usually, the client connection is terminated. The disconnect can result in any of the following options:

1. The master comes back: The connection is restored to the Redis instance
2. A slave is promoted to a master: lettuce performs an address lookup using the `masterId`. As soon as the Redis Sentinel provides an address the connection is restored to the new Redis instance

Read more at http://redis.io/topics/sentinel

5.2.3. Examples

Example 31. Redis Sentinel node connection

```
RedisURI redisUri = RedisURI.create("redis://sentinelhost1:26379");
RedisClient client = new RedisClient(redisUri);

RedisSentinelAsyncConnection<String, String>  connection = client
.connectSentinelAsync();

Map<String, String> map = connection.master("mymaster").get();
```

Example 32. Redis master discovery

```
RedisURI redisUri = RedisURI.Builder.sentinel("sentinelhost1", "mymaster")
.withSentinel("sentinelhost2").build();
RedisClient client = RedisClient.create(redisUri);

RedisConnection<String, String> connection = client.connect();
```

Every time you connect to a Redis instance using Redis Sentinel, the Redis master is looked up using a new connection to a Redis Sentinel. This can be time-consuming, especially when multiple Redis Sentinels are used and one or more of them are not reachable.

5.3. Redis Cluster

lettuce supports Redis Cluster with:

- Support of all `CLUSTER` commands
- Command routing based on the hash slot of the commands' key
- High-level abstraction for selected cluster commands
- Execution of commands on multiple cluster nodes
- `MOVED` and `ASK` redirection handling
- Obtaining direct connections to cluster nodes by slot and host/port (since 3.3)
- SSL and authentication (since 4.2)

- Periodic and adaptive cluster topology updates
- Publish/Subscribe

Connecting to a Redis Cluster requires one or more initial seed nodes. The full cluster topology view (partitions) is obtained on the first connection so you're not required to specify all cluster nodes. Specifying multiple seed nodes helps to improve resiliency as lettuce is able to connect the cluster even if a seed node is not available. Lettuce holds multiple connections, which are opened on demand. You are free to operate on these connections.

Connections can be bound to specific hosts or nodeIds. Connections bound to a nodeId will always stick to the nodeId, even if the nodeId is handled by a different host. Requests to unknown nodeId's or host/ports that are not part of the cluster are rejected. Do not close the connections. Otherwise, unpredictable behavior will occur. Keep also in mind that the node connections are used by the cluster connection itself to perform cluster operations: If you block one connection all other users of the cluster connection might be affected.

5.3.1. Command routing

The concept of Redis Cluster bases on sharding. Every master node within the cluster handles one or more slots. Slots are the unit of sharding and calculated from the commands' key using `CRC16 MOD 16384`. Hash slots can also be specified using hash tags such as `{user:1000}.foo`.

Every request, which incorporates at least one key is routed based on its hash slot to the corresponding node. Commands without a key are executed on the *default* connection that points most likely to the first provided `RedisURI`. The same rule applies to commands operating on multiple keys but with the limitation that all keys have to be in the same slot. Commands operating on multiple slots will be terminated with a `CROSSSLOT` error.

5.3.2. Cross-slot command execution and cluster-wide execution for selected commands

Regular Redis Cluster commands are limited to single-slot keys operation – either single key commands or multi-key commands that share the same hash slot.

The cross slot limitation can be mitigated by using the advanced cluster API for *a set of selected* multi-key commands. Commands that operate on keys with different slots are decomposed into multiple commands. The single commands are fired in a fork/join fashion. The commands are issued concurrently to avoid synchronous chaining. Results are synchronized before the command is completed.

Following commands are supported for cross-slot command execution:

- `DEL`: Delete the `KEY`s. Returns the number of keys that were removed.
- `EXISTS`: Count the number of `KEY`s that exist across the master nodes being responsible for the particular key.
- `MGET`: Get the values of all given `KEY`s. Returns the values in the order of the keys.
- `MSET`: Set multiple key/value pairs for all given `KEY`s. Returns always `OK`.

- `TOUCH`: Alters the last access time of all given `KEY`s. Returns the number of keys that were touched.
- `UNLINK`: Delete the `KEY`s and reclaiming memory in a different thread. Returns the number of keys that were removed.

Following commands are executed on multiple cluster nodes operations:

- `CLIENT SETNAME`: Set the client name on all known cluster node connections. Returns always `OK`.
- `KEYS`: Return/Stream all keys that are stored on all masters.
- `DBSIZE`: Return the number of keys that are stored on all masters.
- `FLUSHALL`: Flush all data on the cluster masters. Returns always `OK`.
- `FLUSHDB`: Flush all data on the cluster masters. Returns always `OK`.
- `RANDOMKEY`: Return a random key from a random master.
- `SCAN`: Scan the keyspace across the whole cluster according to `ReadFrom` settings.
- `SCRIPT FLUSH`: Remove all the scripts from the script cache on all cluster nodes.
- `SCRIPT LOAD`: Load the script into the Lua script cache on all nodes.
- `SCRIPT KILL`: Kill the script currently in execution on all cluster nodes. This call does not fail even if no scripts are running.
- `SHUTDOWN`: Synchronously save the dataset to disk and then shut down all nodes of the cluster.

Cross-slot command execution is available on the following APIs:

- `RedisAdvancedClusterCommands`
- `RedisAdvancedClusterAsyncCommands`
- `RedisAdvancedClusterReactiveCommands`

5.3.3. Execution of commands on one or multiple cluster nodes

Sometimes commands have to be executed on multiple cluster nodes. The advanced cluster API allows to select a set of nodes (e.g. all masters, all slaves) and trigger a command on this set.

Example 33. Using `NodeSelection` API to read all keys from all slaves

```
RedisAdvancedClusterAsyncCommands<String, String> async = clusterClient.connect()
.async();
AsyncNodeSelection<String, String> slaves = connection.slaves();

AsyncExecutions<List<String>> executions = slaves.commands().keys("*");
executions.forEach(result -> result.thenAccept(keys -> System.out.println(keys)));
```

The commands are triggered concurrently. This API is currently only available for async commands. Commands are dispatched to the nodes within the selection, the result (CompletionStage) is available through `AsyncExecutions`.

A node selection can be either dynamic or static. A dynamic node selection updates its node set upon a cluster topology view refresh. Node selections can be constructed by the following presets:

- masters
- slaves (operate on connections with activated `READONLY` mode)
- all nodes

A custom selection of nodes is available by implementing custom predicates or lambdas.

The particular results map to a cluster node (`RedisClusterNode`) that was involved in the node selection. You can obtain the set of involved `RedisClusterNode`s and all results as `CompletableFuture` from `AsyncExecutions`.

The node selection API is a technical preview and can change at any time. That approach allows powerful operations but it requires further feedback from the users. So feel free to contribute.

5.3.4. Refreshing the cluster topology view

The Redis Cluster configuration may change at runtime. New nodes can be added, the master for a specific slot can change. Lettuce handles `MOVED` and `ASK` redirects transparently but in case too many commands run into redirects, you should refresh the cluster topology view. The topology is bound to a `RedisClusterClient` instance. All cluster connections that are created by one `RedisClusterClient` instance share the same cluster topology view. The view can be updated in three ways:

1. Either by calling `RedisClusterClient.reloadPartitions`
2. Periodic updates in the background based on an interval
3. Adaptive updates in the background based on persistent disconnects and `MOVED`/`ASK` redirections

By default, commands follow `-ASK` and `-MOVED` redirects up to 5 times until the command execution is considered to be failed. Background topology updating starts with the first connection obtained through `RedisClusterClient`.

5.3.5. Client-options

See Cluster-specific Client options.

Examples

Example 34. Connecting to a Redis Cluster

```
RedisURI redisUri = RedisURI.Builder.redis("localhost").withPassword(
"authentication").build();

RedisClusterClient clusterClient = RedisClusterClient.create(rediUri);
StatefulRedisClusterConnection<String, String> connection = clusterClient.connect
();
RedisAdvancedClusterCommands<String, String> syncCommands = connection.sync();

...

connection.close();
clusterClient.shutdown();
```

Example 35. Connecting to a Redis Cluster with multiple seed nodes

```
RedisURI node1 = RedisURI.create("node1", 6379);
RedisURI node2 = RedisURI.create("node2", 6379);

RedisClusterClient clusterClient = RedisClusterClient.create(Arrays.asList(node1,
node2));
StatefulRedisClusterConnection<String, String> connection = clusterClient.connect
();
RedisAdvancedClusterCommands<String, String> syncCommands = connection.sync();

...

connection.close();
clusterClient.shutdown();
```

Example 36. Enabling periodic cluster topology view updates

```
RedisClusterClient clusterClient = RedisClusterClient.create(RedisURI.create(
"localhost", 6379));

ClusterTopologyRefreshOptions topologyRefreshOptions =
ClusterTopologyRefreshOptions.builder()
                                .enablePeriodicRefresh(10, TimeUnit.MINUTES)
                                .build();

clusterClient.setOptions(ClusterClientOptions.builder()
                                .topologyRefreshOptions(topologyRefreshOptions)
                                .build());
...

clusterClient.shutdown();
```

Example 37. Enabling adaptive cluster topology view updates

```
RedisURI node1 = RedisURI.create("node1", 6379);
RedisURI node2 = RedisURI.create("node2", 6379);

RedisClusterClient clusterClient = RedisClusterClient.create(Arrays.asList(node1,
node2));

ClusterTopologyRefreshOptions topologyRefreshOptions =
ClusterTopologyRefreshOptions.builder()
                                .enableAdaptiveRefreshTrigger(RefreshTrigger
.MOVED_REDIRECT, RefreshTrigger.PERSISTENT_RECONNECTS)
                                .adaptiveRefreshTriggersTimeout(30, TimeUnit
.SECONDS)
                                .build();

clusterClient.setOptions(ClusterClientOptions.builder()
                                .topologyRefreshOptions(topologyRefreshOptions)
                                .build());
...

clusterClient.shutdown();
```

Example 38. Obtaining a node connection

```
RedisURI node1 = RedisURI.create("node1", 6379);
RedisURI node2 = RedisURI.create("node2", 6379);

RedisClusterClient clusterClient = RedisClusterClient.create(Arrays.asList(node1,
node2));
StatefulRedisClusterConnection<String, String> connection = clusterClient.connect
();

RedisClusterCommands<String, String> node1 = connection.getConnection("host",
7379).sync();

...
// do not close node1

connection.close();
clusterClient.shutdown();
```

5.4. ReadFrom Settings

The ReadFrom setting describes how lettuce routes read operations to slave nodes.

By default, lettuce routes its read operations in multi-node connections to the master node. Reading from the master returns the most recent version of the data because write operations are issued to the single master node. Reading from masters guarantees strong consistency.

You can reduce latency or improve read throughput by distributing reads to slave members for applications that do not require fully up-to-date data.

Be careful if using other ReadFrom settings than `MASTER`. Settings other than `MASTER` may return stale data because the replication is asynchronous. Data in the slaves may not hold the most recent data.

5.4.1. Redis Cluster

Redis Cluster is a multi-node operated Redis setup that uses one or more master nodes and allows to setup slave nodes. Redis Cluster connections allow to set a `ReadFrom` setting on connection level. This setting applies for all read operations on this connection.

Example 39. Enable Slave reads with `ReadFrom.SLAVE`

```
RedisClusterClient client = RedisClusterClient.create(RedisURI.create("host",
7379));
StatefulRedisClusterConnection<String, String> connection = client.connect();
connection.setReadFrom(ReadFrom.SLAVE);

RedisAdvancedClusterCommands<String, String> sync = connection.sync();
sync.set(key, "value");

sync.get(key); // slave read

connection.close();
client.shutdown();
```

5.4.2. Master/Slave connections

Redis nodes can be operated in a Master/Slave setup to achieve availability and performance. Master/Slave setups can be run either Standalone or managed using Redis Sentinel. Lettuce allows to use slave nodes for read operations by using the `MasterSlave` API that supports both Master/Slave setups:

1. Redis Standalone Master/Slave (no failover)
2. Redis Sentinel Master/Slave (Sentinel-managed failover)

The resulting connection uses in any case the primary connection-point to dispatch non-read operations.

Redis Sentinel

Master/Slave with Redis Sentinel is very similar to regular Redis Sentinel operations. When the master fails over, a slave is promoted by Redis Sentinel to the new master and the client obtains the new topology from Redis Sentinel.

Connections to Master/Slave require one or more Redis Sentinel connection points and a master name. The primary connection point is the Sentinel monitored master node.

Example 40. Using ReadFrom with Master/Slave and Redis Sentinel

```
RedisURI sentinelUri = RedisURI.Builder.sentinel("sentinel-host", 26379, "master-
name").build();
RedisClient client = RedisClient.create();

StatefulRedisMasterSlaveConnection<String, String> connection = MasterSlave
.connect(
            client,
            new Utf8StringCodec(),
            sentinelUri);

connection.setReadFrom(ReadFrom.SLAVE);

connection.sync().get("key"); // Slave read

connection.close();
client.shutdown();
```

Redis Standalone

Master/Slave with Redis Standalone is very similar to regular Redis Standalone operations. A Redis Standalone Master/Slave setup is static and provides no built-in failover. Slaves are read from the Redis Master node's `INFO` command.

Connecting to Redis Standalone Master/Slave nodes requires connections to use the Redis Master for the `RedisURI`. The node used within the `RedisURI` is the primary connection point.

Example 41. Using ReadFrom with Master/Slave and Redis Standalone (Master and Slave)

```
RedisURI masterUri = RedisURI.Builder.redis("master-host", 6379).build();
RedisClient client = RedisClient.create();

StatefulRedisMasterSlaveConnection<String, String> connection = MasterSlave
.connect(
            client,
            new Utf8StringCodec(),
            masterUri);

connection.setReadFrom(ReadFrom.SLAVE);

connection.sync().get("key"); // Slave read

connection.close();
client.shutdown();
```

5.4.3. Use Cases for non-master reads

The following use cases are common for using non-master read settings and encourage eventual consistency:

- Providing local reads for geographically distributed applications. If you have Redis and application servers in multiple data centers, you may consider having a geographically distributed cluster. Using the `NEAREST` setting allows the client to read from the lowest-latency members, rather than always reading from the master node.
- Maintaining availability during a failover. Use `MASTER_PREFERRED` if you want an application to read from the master by default, but to allow stale reads from slaves when the master node is unavailable. `MASTER_PREFERRED` allows a "read-only mode" for your application during a failover.
- Increase read throughput by allowing stale reads If you want to increase your read throughput by adding additional slave nodes to your cluster Use `SLAVE` to read explicitly from slaves and reduce read load on the master node. Using slave reads can highly lead to stale reads.

5.4.4. Read from settings

All `ReadFrom` settings except `MASTER` may return stale data because slaves replication is asynchronous and requires some delay. You need to ensure that your application can tolerate stale data.

Setting	Description
`MASTER`	Default mode. Read from the current master node.
`MASTER_PREFERRED`	Read from the master, but if it is unavailable, read from slave nodes.
`SLAVE`	Read from slave nodes.
`NEAREST`	Read from any node of the cluster with the lowest latency.

The latency of the nodes is determined upon cluster topology refresh. If the topology view is never refreshed, values from the initial cluster nodes read are used.

Custom read settings can be implemented by extending the `com.lambdaworks.redis.ReadFrom` class.

Chapter 6. Working with dynamic Redis Command Interfaces

The Redis Command Interface abstraction provides a dynamic way for typesafe Redis command invocation. It allows you to declare an interface with command methods to significantly reduce boilerplate code required to invoke a Redis command.

6.1. Introduction

Redis is a data store supporting over 190 documented commands and over 450 command permutations. The community supports actively Redis development; each major Redis release comes with new commands. Command growth and keeping track with upcoming modules are challenging for client developers and Redis user as there is no full command coverage for each module in a single Redis client.

The central interface in lettuce Command Interface abstraction is `Commands`. This interface acts primarily as a marker interface to help you to discover interfaces that extend this one. The `KeyCommands` interface below declares some command methods.

Example 42. Command interface

```
public interface KeyCommands extends Commands {

    String get(String key);                 ①

    String set(String key, String value);   ②

    String set(String key, byte[] value);   ③
}
```

① Retrieves a key by its name.

② Sets a key and value.

③ Sets a key and a value by using bytes.

The interface from above declares several methods. Let's take a brief look at `String set(String key, String value)`. We can derive from that declaration certain things:

- It should be executed synchronously – there's no asynchronous or reactive wrapper declared in the result type.
- The Redis command method returns a `String` - that reveals something regarding the command result expectation. This command expects a reply that can be represented as `String`.
- The method is named `set` so the derived command will be named `set`.
- There are two parameters defined: `String key` and `String value`. Although Redis does not take any other parameter types than bulk strings, we still can apply a transformation to the

parameters – we can conclude their serialization from the declared type.

The `set` command from above called would look like:

```
commands.set("key", "value");
```

This command translates to:

```
SET key value
```

6.2. Command methods

With lettuce, declaring command methods becomes a four-step process:

1. Declare an interface extending `Commands`.

   ```
   interface KeyCommands extends Commands { ⋯ }
   ```

2. Declare command methods on the interface.

   ```
   interface KeyCommands extends Commands {
       String get(String key);
   }
   ```

3. Set up lettuce to create proxy instances for those interfaces.

   ```
   RedisClient client = ⋯
   RedisCommandFactory factory = new RedisCommandFactory(client.connect());
   ```

4. Get the commands instance and use it.

   ```
   public class SomeClient {

       KeyCommands commands;

       public SomeClient(RedisCommandFactory factory) {
           commands = factory.getCommands(KeyCommands.class);
       }

       public void doSomething() {
           String value = commands.get("Walter");
       }
   }
   ```

The sections that follow explain each step in detail.

6.3. Defining command methods

As a first step, you define a specific command interface. The interface must extend `Commands`.

Command methods are declared inside the commands interface like regular methods (probably not that much of a surprise). Lettuce derives commands (name, arguments, and response) from each declared method.

6.3.1. Command naming

The commands proxy has two ways to derive a Redis command from the method name. It can derive the command name from the method name directly, or by using a manually defined `@Command` annotation. However, there's got to be a strategy that decides what actual command is created. Let's have a look at the available options.

Example 43. `MixedCommands` interface annotated with `@Command` and `@CommandNaming`

```
public interface MixedCommands extends Commands {

    List<String> mget(String... keys);                  ①

    @Command("MGET")
    List<Value<String> mgetAsValues(String... keys);    ②

    @CommandNaming(strategy = DOT)
    double nrRun(String key, int... indexes)            ③
}
```

① Plain command method. Lettuce will derive to the `MGET` command.

② Command method annotated with `@Command`. Lettuce will execute `MGET` since annotations have a higher precedence than method-based name derivation.

③ Redis commands consist of one or multiple command parts or follow a different naming strategy. The recommended pattern for commands provided by modules is using dot notation. Command methods can derive from "camel humps" that style by placing a dot (`.`) between name parts.

6.3.2. CamelCase in method names

Command methods use by default the method name command type. This is ideal for commands like `GET`, `SET`, `ZADD` and so on. Some commands, such as `CLIENT SETNAME` consist of multiple command segments and passing `SETNAME` as argument to a method `client(…)` feels rather clunky.

Camel case is a natural way to express word boundaries in method names. These "camel humps" (changes in letter casing) can be interpreted in different ways. The most common case is to translate a change in case into a space between command segments.

```
interface ServerCommands extends Commands {
    String clientSetname(String name);
}
```

Invoking `clientSetname(…)` will execute the Redis command `CLIENT SETNAME name`.

`@CommandNaming`

Camel humps are translated to whitespace-delimited command segments by default. Methods and the commands interface can be annotated with `@CommandNaming` to apply a different strategy.

```
@CommandNaming(strategy = Strategy.DOT)
interface MixedCommands extends Commands {

    @CommandNaming(strategy = Strategy.SPLIT)
    String clientSetname(String name);

    @CommandNaming(strategy = Strategy.METHOD_NAME)
    String mSet(String key1, String value1, String key2, String value2);

    double nrRun(String key, int... indexes)
}
```

You can choose amongst multiple strategies:

- `SPLIT`: Splits camel-case method names into multiple command segments: `clientSetname` executes `CLIENT SETNAME`. This is the default strategy.
- `METHOD_NAME`: Uses the method name as-is: `mSet` executes `MSET`.
- `DOT`: Translates camel-case method names into dot-notation that is the recommended pattern for module-provided commands. `nrRun` executes `NR.RUN`.

6.3.3. `@Command` annotation

You already learned, that method names are used as command type any by default all arguments are appended to the command. Some cases, such as the example from above, require in Java declaring a method with a different name because of variance in the return type. `mgetAsValues` would execute a non-existent command `MGETASVALUES`.

Annotating command methods with `@Command` lets you take control over implicit conventions. The annotation value overrides the command name and provides command segments to command methods. Command segments are parts of a command that are sent to Redis. The semantics of a command segment depend on context and the command itself. `@Command("CLIENT SETNAME")` denotes a subcommand of the `CLIENT` command while a method annotated with `@Command("SET key")` invokes `SET`, using `mykey` as key. `@Command` lets you specify whole command strings and reference parameters to construct custom commands.

```
interface MixedCommands extends Commands {

    @Command("CLIENT SETNAME")
    String setName(String name);

    @Command("MGET")
    List<Value<String> mgetAsValues(String... keys);

    @Command("SET mykey")
    String set(String value);

    @Command("NR.OBSERVE ?0 ?1 -> ?2 TRAIN")
    List<Integer> nrObserve(String key, int[] in, int... out)
}
```

6.3.4. Parameters

Most Redis commands take one or more parameters to operate with your data. Using command methods with Redis appends all parameters in their specified order to the command as arguments. You have already seen commands annotated with `@Command("MGET")` or with no annotation at all. Commands append their parameters as command arguments as declared in the method signature.

```
interface MixedCommands extends Commands {

    @Command("SET ?1 ?0")
    String set(String value, String key);

    @Command("NR.OBSERVE :key :in -> :out TRAIN")
    List<Integer> nrObserve(@Param("key") String key, @Param("in") int[] in, @Param(
"out") int... out)
}
```

`@Command`-annotated command methods allow references to parameters. You can use index-based or name-based parameter references. Index-based references (`?0`, `?1`, …) are zero-based. Name-based parameters (`:key`, `:in`) reference parameters by their name. Java 8 provides access to parameter names if the code was compiled with `javac -parameters`. Parameter names can be supplied alternatively by `@Param`. Please note that all parameters are required to be annotated if using `@Param`.

The same parameter can be referenced multiple times. Not referenced parameters are appended as arguments after the last command segment.

Keys and values

Redis commands are usually less concerned about key and value type since all data is bytes anyway. In the context of Redis Cluster, the very first key affects command routing. Keys and values are discovered by verifying their declared type assignability to `RedisCodec` key and value types. In some cases, where keys and values are indistinguishable from their types, it might be required to hint

command methods about keys and values. You can annotate key and value parameters with `@Key` and `@Value` to control which parameters should be treated as keys or values.

```
interface KeyCommands extends Commands {

    String set(@Key String key, @Value String value);
}
```

Hinting command method parameters influences `RedisCodec` selection.

Parameter types

Command method parameter types are just limited by the `RedisCodecs` that are supplied to `RedisCommandFactory`. Command methods, however, support a basic set of parameter types that are agnostic to the selected codec. If a parameter is identified as key or value and the codec supports that parameter, this specific parameter is encoded by applying codec conversion.

Built-in parameter types:

- `String` - encoded to bytes using `ASCII`.
- `byte[]`
- `double`/`Double`
- `ProtocolKeyword` - using its byte-representation. `ProtocolKeyword` is useful to declare/reuse commonly used Redis keywords, see `com.lambdaworks.redis.protocol.CommandType` and `com.lambdaworks.redis.protocol.CommandKeyword`.
- `Map` - key and value encoding of key-value pairs using `RedisCodec`.
- types implementing `com.lambdaworks.redis.CompositeParameter` - Lettuce comes with a set of command argument types such as `BitFieldArgs`, `SetArgs`, `SortArgs`, … that can be used as parameter. Providing `CompositeParameter` will ontribute multiple command arguments by invoking the `CompositeParameter.build(CommandArgs)` method.
- `Value`, `KeyValue`, and `ScoredValue` that are encoded to their value, key and value and score and value representation using `RedisCodec`.
- `GeoCoordinates` - contribute longitude and latitude command arguments
- `Limit` - used together with `ZRANGEBYLEX`/`ZRANGEBYSCORE` commands. Will add `LIMIT (offset) (count)` segments to the command.
- `Range` - used together with `ZCOUNT`/`ZRANGEBYLEX`/`ZRANGEBYSCORE` commands. Numerical commands are converted to numerical boundaries (`+inf`, `(1.0`, `[1.0`). Value-typed `Range` parameters are encoded to their value boundary representation (`+`, `-`, `[value`, `(value`).

Command methods accept other, special parameter types such as `Timeout` or `FlushMode` that control execution-model specific behavior. Those parameters are filtered from command arguments.

6.3.5. Codecs

Redis command interfaces use `RedisCodecs` for key/value encoding and decoding. Each command

method performs RedisCodec resolution so each command method can use a different RedisCodec. Codec resolution is based on key and value types declared in the command method signature. Key and value parameters can be annotated with @Key/@Value annotations to hint codec resolution to the appropriate types. Codec resolution checks all annotated parameters for compatibility. If types are assignable to codec types, the codec is selected for a particular command method.

Codec resolution without annotation is based on compatible type majority. A command method resolves to the codec accepting the most compatible types. See also Keys and values for details on key/value encoding. Depending on provided codecs and the command method signature it's possible that no codec can be resolved. You need to provide either a compatible RedisCodec or adjust parameter types in the method signature to provide a compatible method signature. RedisCommandFactory uses StringCodec (UTF-8) and ByteArrayCodec by default.

Example 44. Initialize RedisCommandFactory with multiple RedisCodecs

```
RedisCommandFactory factory = new RedisCommandFactory(connection, Arrays.asList
(new ByteArrayCodec(), new StringCodec(LettuceCharsets.UTF8)));
```

The resolved codec is also applied to command response deserialization that allows you to use parametrized command response types.

6.3.6. Response types

Another aspect of command methods is their response type. Redis command responses consist of simple strings, bulk strings (byte streams) or arrays with nested elements depending on the issued command.

You can choose amongst various return types that map to a particular {custom-commands-command-output-link}. A command output can return either its return type directly (List<String> for StringListOutput) or stream individual elements (String for StringListOutput as it implements StreamingOutput<String>). Command output resolution depends on whether the declared return type supports streaming. The currently only supported streaming output are reactive wrappers such as Flux.

RedisCommandFactory comes with built-in command outputs that are resolved from OutputRegistry. You can choose from built-in command output types or register your own CommandOutput.

A command method can return its response directly or wrapped in a response wrapper. See Execution models for execution-specific wrapper types.

Table 1. Built-in command output types

CommandOutput class	return type	streaming type
ListOfMapsOutput	List<Map<K, V>>	
ArrayOutput	List<Object>	
DoubleOutput	Double, double	
ByteArrayOutput	byte[]	

`CommandOutput` class	return type	streaming type
`IntegerOutput`	`Long`, `long`	
`KeyOutput`	`K` (Codec key type)	
`KeyListOutput`	`List<K>` (Codec key type)	`K` (Codec key type)
`ValueOutput`	`V` (Codec value type)	
`ValueListOutput`	`List<V>` (Codec value type)	`V` (Codec value type)
`ValueSetOutput`	`Set<V>` (Codec value type)	
`MapOutput`	`Map<K, V>`	
`BooleanOutput`	`Boolean`, `boolean`	
`BooleanListOutput`	`List<Boolean>`	`Boolean`
`GeoCoordinatesListOutput`	`GeoCoordinates`	
`GeoCoordinatesValueListOutput`	`List<Value<GeoCoordinates>>`	`Value<GeoCoordinates>`
`ScoredValueListOutput`	`List<ScoredValue<V>>`	`ScoredValue<V>`
`StringValueListOutput` (ASCII)	`List<Value<String>>`	`Value<String>`
`StringListOutput` (ASCII)	`List<String>`	`String`
`ValueValueListOutput`	`List<Value<V>>`	`Value<V>`
`VoidOutput`	`Void`, `void`	

6.4. Execution models

Each declared command methods requires a synchronization mode, more specific an execution model. Lettuce uses an event-driven command execution model to send commands, process responses, and signal completion. Command methods can execute their commands in a synchronous, asynchronous or reactive way.

The choice of a particular execution model is made on return type level, more specific on the return type wrapper. Each command method may use a different execution model so command methods within a command interface may mix different execution models.

6.4.1. Synchronous (Blocking) Execution

Declaring a non-wrapped return type (like `List<V>`, `String`) will execute commands synchronously. See {custom-commands-command-exec-model-link} on more details on synchronous command execution.

Blocking command execution applies by default timeouts set on connection level. Command methods support timeouts on invocation level by defining a special `Timeout` parameter. The parameter position does not affect command segments since special parameters are filtered from the command arguments. Supplying `null` will apply connection defaults.

```
interface KeyCommands extends Commands {

    String get(String key, Timeout timeout);
}

KeyCommands commands = ⋯

commands.get("key", Timeout.create(10, TimeUnit.SECONDS));
```

6.4.2. Asynchronous (Future) Execution

Command methods wrapping their response in `Future`, `CompletableFuture`, `CompletionStage` or `RedisFuture` will execute their commands asynchronously. Invoking an asynchronous command method will send the command to Redis at invocation time and return a return handle that allows you to synchronize or chain command execution.

```
interface KeyCommands extends Commands {

    RedisFuture<String> get(String key, Timeout timeout);
}
```

6.4.3. Reactive Execution

You can declare command methods that wrap their response in a reactive type for reactive command execution. Invoking a reactive command method will not send the command to Redis until the resulting subscriber signals demand for data to its subscription. Using reactive wrapper types allow result streaming by emitting data as it's received from the I/O channel.

Currently supported reactive types:

- Project Reactor `Mono` and `Flux` (native)
- RxJava 1 `Single` and `Observable` (via `rxjava-reactive-streams`)
- RxJava 2 `Single`, `Maybe` and `Flowable` (via `rxjava` 2.0)

See Reactive API for more details.

```
interface KeyCommands extends Commands {

    @Command("GET")
    Mono<String> get(String key);

    @Command("GET")
    Maybe<String> getRxJava2Maybe(String key);

    Flowable<String> lrange(String key, long start, long stop);
}
```

6.4.4. Batch Execution

Command interfaces support command batching to collect multiple commands in a batch queue and flush the batch in a single write to the transport. Command batching executes commands in a deferred nature. This means that at the time of invocation no result is available. Batching can be only used with synchronous methods without a return value (`void`) or asynchronous methods returning a `RedisFuture`. Reactive command batching is not supported because reactive executed commands maintain an own subscription lifecycle that is decoupled from command method batching.

Command batching can be enabled on two levels:

- On class level by annotating the command interface with `@BatchSize`. All methods participate in command batching.
- On method level by adding `CommandBatching` to the arguments. Method participates selectively in command batching.

```
@BatchSize(50)
interface StringCommands extends Commands {

    void set(String key, String value);

    RedisFuture<String> get(String key);

    RedisFuture<String> get(String key, CommandBatching batching);
}

StringCommands commands = ⋯

commands.set("key", "value"); // queued until 50 command invocations reached.
                              // The 50th invocation flushes the queue.

commands.get("key", CommandBatching.queue()); // invocation-level queueing control
commands.get("key", CommandBatching.flush()); // invocation-level queueing control,
                                              // flushes all queued commands
```

Batching can be controlled on per invocation by passing a `CommandBatching` argument. `CommandBatching` has precedence over `@BatchSize`.

To flush queued commands at any time (without further command invocation), add `BatchExecutor` to your interface definition.

```
@BatchSize(50)
interface StringCommands extends Commands, BatchExecutor {

    RedisFuture<String> get(String key);
}

StringCommands commands = ⋯

commands.set("key");

commands.flush() // force-flush
```

Batch execution synchronization

Queued command batches are flushed either on reaching the batch size or force flush (via `BatchExecutor.flush()` or `CommandBatching.flush()`). Errors are transported through `RedisFuture`. Synchronous commands don't receive any result/exception signal except if the batch is flushed through a synchronous method call. Synchronous flushing throws `BatchException` containing the failed commands.

Chapter 7. Advanced usage

7.1. Configuring Client resources

Client resources are configuration settings for the client related to performance, concurrency, and events. A vast part of Client resources consists of thread pools (`EventLoopGroup`s and a `EventExecutorGroup`) which build the infrastructure for the connection workers. In general, it is a good idea to reuse instances of `ClientResources` across multiple clients.

Client resources are available since 3.4 and 4.1.

Client resources are stateful and need to be shut down if they are supplied from outside the client.

7.1.1. Creating Client resources

Client resources are required to be immutable. You can create instances using two different patterns:

The `create()` factory method

By using the `create()` method on `DefaultClientResources` you create `ClientResources` with default settings:

```
ClientResources res = DefaultClientResources.create();
```

This approach fits the most needs.

Resources builder

You can build instances of `DefaultClientResources` by using the embedded builder. It is designed to configure the resources to your needs. The builder accepts the configuration in a fluent fashion and then creates the ClientResources at the end:

```
ClientResources res = DefaultClientResources.builder()
                        .ioThreadPoolSize(4)
                        .computationThreadPoolSize(4)
                        .build()
```

7.1.2. Using and reusing `ClientResources`

A `RedisClient` and `RedisClusterClient` can be created without passing `ClientResources` upon creation. The resources are exclusive to the client and are managed itself by the client. When calling `shutdown()` of the client instance `ClientResources` are shut down.

```
RedisClient client = RedisClient.create();
...
client.shutdown();
```

If you require multiple instances of a client or you want to provide existing thread infrastructure, you can configure a shared `ClientResources` instance using the builder. The shared Client resources can be passed upon client creation:

```
ClientResources res = DefaultClientResources.create();
RedisClient client = RedisClient.create(res);
RedisClusterClient clusterClient = RedisClusterClient.create(res, seedUris);
...
client.shutdown();
clusterClient.shutdown();
res.shutdown();
```

Shared `ClientResources` are never shut down by the client. Same applies for shared `EventLoopGroupProvider`s that are an abstraction to provide `EventLoopGroup`s.

Why `Runtime.getRuntime().availableProcessors()` * 3?

Netty requires different `EventLoopGroup`s for NIO (TCP) and for EPoll (Unix Domain Socket) connections. One additional `EventExecutorGroup` is used to perform computation tasks. `EventLoopGroup`s are started lazily to allocate Threads on-demand.

Shutdown

Every client instance requires a call to `shutdown()` to clear used resources. Clients with dedicated `ClientResources` (i.e. no `ClientResources` passed within the constructor/`create`-method) will shut down `ClientResources` on their own.

Client instances with using shared `ClientResources` (i.e. `ClientResources` passed using the constructor/`create`-method) won't shut down the `ClientResources` on their own. The `ClientResources` instance needs to be shut down once its not used anymore.

7.1.3. Configuration settings

The basic configuration options are listed in the table below:

Name	Method	Default
I/O Thread Pool Size	`ioThreadPoolSize`	`Number of processors`
The number of threads in the I/O thread pools. The number defaults to the number of available processors that the runtime returns (which, as a well-known fact, sometimes does not represent the actual number of processors). Every thread represents an internal event loop where all I/O tasks are run. The number does not reflect the actual number of I/O threads because the client requires different thread pools for Network (NIO) and Unix Domain Socket (EPoll) connections. The minimum I/O threads are `3`. A pool with fewer threads can cause undefined behavior.		

Name	Method	Default
Computation Thread Pool Size	`computationThreadPoolSize`	`Number of processors`
The number of threads in the computation thread pool. The number defaults to the number of available processors that the runtime returns (which, as a well-known fact, sometimes does not represent the actual number of processors). Every thread represents an internal event loop where all computation tasks are run. The minimum computation threads are `3`. A pool with fewer threads can cause undefined behavior.		

7.1.4. Advanced settings

Values for the advanced options are listed in the table below and should not be changed unless there is a truly good reason to do so.

Name	Method	Default
Provider for EventLoopGroup	`eventLoopGroupProvider`	`none`
For those who want to reuse existing netty infrastructure or the total control over the thread pools, the `EventLoopGroupProvider` API provides a way to do so. `EventLoopGroup`s are obtained and managed by an `EventLoopGroupProvider`. A provided `EventLoopGroupProvider` is not managed by the client and needs to be shutdown once you do not longer need the resources.		
Provided EventExecutorGroup	`eventExecutorGroup`	`none`
For those who want to reuse existing netty infrastructure or the total control over the thread pools can provide an existing `EventExecutorGroup` to the Client resources. A provided `EventExecutorGroup` is not managed by the client and needs to be shutdown once you do not longer need the resources.		
Event bus	`eventBus`	`DefaultEventBus`
The event bus system is used to transport events from the client to subscribers. Events are about connection state changes, metrics, and more. Events are published using a RxJava subject and the default implementation drops events on backpressure. Learn more about the link:Reactive API (4.0)[Reactive API]. You can also publish your own events. If you wish to do so, make sure that your events implement the `Event` marker interface.		
Command latency collector options	`commandLatencyCollectorOptions`	`DefaultCommandLatencyCollectorOptions`
The client can collect latency metrics during while dispatching commands. The options allow to configure the percentiles, level of metrics (per connection or server) and whether the metrics are cumulative or reset after obtaining these. Command latency collection is enabled by default and can be disabled by setting `commandLatencyPublisherOptions(…)` to `DefaultEventPublisherOptions.disabled()`. Latency collector requires `LatencyUtils` to be on your class path.		
Command latency collector	`commandLatencyCollector`	`DefaultCommandLatencyCollector`
The client can collect latency metrics during while dispatching commands. Command latency metrics is collected on connection or server level. Command latency collection is enabled by default and can be disabled by setting `commandLatencyCollectorOptions(…)` to `DefaultCommandLatencyCollectorOptions.disabled()`.		
Latency event publisher options	`commandLatencyPublisherOptions`	`DefaultEventPublisherOptions`

Name	Method	Default
Command latencies can be published using the event bus. Latency events are emitted by default every 10 minutes. Event publishing can be disabled by setting `commandLatencyPublisherOptions(…)` to `DefaultEventPublisherOptions.disabled()`.		
DNS Resolver	`dnsResolver`	`DnsResolvers.JVM_DEFAULT (or netty if present)`
Since: 3.5, 4.2 Configures a DNS resolver to resolve hostnames to a `java.net.InetAddress`. Defaults to the JVM DNS resolution that uses blocking hostname resolution and caching of lookup results. Users of DNS-based Redis-HA setups (e.g. AWS ElastiCache) might want to configure a different DNS resolver. Lettuce comes with `DirContextDnsResolver` that uses Java's `DnsContextFactory` to resolve hostnames. `DirContextDnsResolver` allows using either the system DNS or custom DNS servers without caching of results so each hostname lookup yields in a DNS lookup. Since 4.4: Defaults to `DnsResolvers.UNRESOLVED` to use netty's `AddressResolver` that resolves DNS names on `Bootstrap.connect()` (requires netty 4.1)		
Reconnect Delay	`reconnectDelay`	`Delay.exponential()`
Since: 4.2 Configures a reconnect delay used to delay reconnect attempts. Defaults to binary exponential delay with an upper boundary of `30 SECONDS`. See `Delay` for more delay implementations.		
Netty Customizer	`NettyCustomizer`	`none`
Since: 4.4 Configures a netty customizer to enhance netty components. Allows customization of `Bootstrap` after `Bootstrap` configuration by Lettuce and `Channel` customization after all Lettuce handlers are added to `Channel`. The customizer allows custom SSL configuration (requires RedisURI in plain-text mode, otherwise Lettuce's configures SSL), adding custom handlers or setting customized `Bootstrap` options. Misconfiguring `Bootstrap` or `Channel` can cause connection failures or undesired behavior.		

7.2. Client Options

Client options allow controlling behavior for some specific features.

Client options are immutable. Connections inherit the current options at the moment the connection is created. Changes to options will not affect existing connections.

```
client.setOptions(ClientOptions.builder()
                        .autoReconnect(false)
                        .pingBeforeActivateConnection(true)
                        .build());
```

Name	Method	Default
PING before activating connection	`pingBeforeActivateConnection`	`false`

Name	Method	Default
Since: 3.1, 4.0 Enables initial `PING` barrier before any connection is usable. If `true`, every connection and reconnect will issue a `PING` command and awaits its response before the connection is activated and enabled for use. If the check fails, the connect/reconnect is treated as a failure. Failed `PING`s on reconnect are handled as protocol errors and can suspend reconnection if `suspendReconnectOnProtocolFailure` is enabled. The `PING` will validate whether the other end of the connected socket is a service that behaves like a Redis server.		
Auto-Reconnect	`autoReconnect`	`true`
Since: 3.1, 4.0 Controls auto-reconnect behavior on connections. As soon as a connection gets closed/reset without the intention to close it, the client will try to reconnect, activate the connection and re-issue any queued commands. This flag also has the effect that disconnected connections will refuse commands and cancel these with an exception.		
Cancel commands on reconnect failure	`cancelCommandsOnReconnectFailure`	`false`
Since: 3.1, 4.0 If this flag is `true` any queued commands will be canceled when a reconnect fails within the activation sequence. The reconnect itself has two phases: Socket connection and protocol/connection activation. In case a connect timeout occurs, a connection reset, host lookup fails, this does not affect the cancelation of commands. In contrast, where the protocol/connection activation fails due to SSL errors or PING before activating connection failure, queued commands are canceled.		
Suspend reconnect on protocol failure	`suspendReconnectOnProtocolFailure`	`false (was introduced in 3.1 with default true)`
Since: 3.1, 4.0 If this flag is `true` the reconnect will be suspended on protocol errors. The reconnect itself has two phases: Socket connection and protocol/connection activation. In case a connect timeout occurs, a connection reset, host lookup fails, this does not affect the cancellation of commands. In contrast, where the protocol/connection activation fails due to SSL errors or PING before activating connection failure, queued commands are canceled. Reconnection can be activated again, but there is no public API to obtain the `ConnectionWatchdog` instance.		
Request queue size	`requestQueueSize`	`2147483647 (Integer#MAX_VALUE)`

Name	Method	Default
Since: 3.4, 4.1 Controls the per-connection request queue size. The command invocation will lead to a `RedisException` if the queue size is exceeded. Setting the `requestQueueSize` to a lower value will lead earlier to exceptions during overload or while the connection is in a disconnected state. A higher value means hitting the boundary will take longer to occur, but more requests will potentially be queued, and more heap space is used.		
Disconnected behavior	`disconnectedBehavior`	`DEFAULT`
Since: 3.4, 4.1 A connection can behave in a disconnected state in various ways. The auto-connect feature allows in particular to retrigger commands that have been queued while a connection is disconnected. The disconnected behavior setting allows fine-grained control over the behavior. Following settings are available: `DEFAULT`: Accept commands when auto-reconnect is enabled, reject commands when auto-reconnect is disabled. `ACCEPT_COMMANDS`: Accept commands in disconnected state. `REJECT_COMMANDS`: Reject commands in disconnected state.		
SSL Options	`sslOptions`	`(none), use JDK defaults`
Since: 4.3 Configure SSL options regarding SSL providers (JDK/OpenSSL) and key store/trust store.		
Socket Options	`socketOptions`	`10 seconds Connection-Timeout, no keep-alive, no TCP noDelay`
Since: 4.3 Options to configure low-level socket options for the connections kept to Redis servers.		
Timeout Options	`timeoutOptions`	`Do not timeout commands.`
Since: 5.1 Options to configure command timeouts applied to timeout commands after dispatching these (active connections, queued while disconnected, batch buffer). By default, use synchronization timeouts only on the synchronous API.		

7.2.1. Cluster-specific options

Cluster client options extend the regular client options by some cluster specifics.

Cluster client options are immutable. Connections inherit the current options at the moment the connection is created. Changes to options will not affect existing connections.

```
ClusterTopologyRefreshOptions topologyRefreshOptions = ClusterTopologyRefreshOptions
.builder()
                .enablePeriodicRefresh(refreshPeriod(10, TimeUnit.MINUTES))
                .enableAllAdaptiveRefreshTriggers()
                .build();

client.setOptions(ClusterClientOptions.builder()
                      .topologyRefreshOptions(topologyRefreshOptions)
                      .build());
```

Name	Method	Default
Periodic cluster topology refresh	`enablePeriodicRefresh`	`false`
Since: 3.1, 4.0 Enables or disables periodic cluster topology refresh. The refresh is handled in the background. Partitions, the view on the Redis cluster topology, are valid for a whole `RedisClusterClient` instance, not a connection. All connections created by this client operate on the one cluster topology. The refresh job is regularly executed, the period between the runs can be set with `refreshPeriod`. The refresh job starts after either opening the first connection with the job enabled or by calling `reloadPartitions`. The job can be disabled without discarding the full client by setting new client options.		
Cluster topology refresh period	`refreshPeriod`	`60 SECONDS`
Since: 3.1, 4.0 Set the period between the refresh job runs. The effective interval cannot be changed once the refresh job is active. Changes to the value will be ignored.		
Adaptive cluster topology refresh	`enableAdaptiveRefreshTrigger`	`(none)`
Since: 4.2 Enables selectively adaptive topology refresh triggers. Adaptive refresh triggers initiate topology view updates based on events happened during Redis Cluster operations. Adaptive triggers lead to an immediate topology refresh. These refreshes are rate-limited using a timeout since events can happen on a large scale. Adaptive refresh triggers are disabled by default. Following triggers can be enabled: `MOVED_REDIRECT`, `ASK_REDIRECT`, `PERSISTENT_RECONNECTS` (see also reconnect attempts for the reconnect trigger)		
Adaptive refresh triggers timeout	`adaptiveRefreshTriggersTimeout`	`30 SECONDS`

Name	Method	Default
Since: 4.2 Set the timeout between the adaptive refresh job runs. Multiple triggers within the timeout will be ignored, only the first enabled trigger leads to a topology refresh. The effective period cannot be changed once the refresh job is active. Changes to the value will be ignored.		
Reconnect attempts (Adaptive topology refresh trigger)	`refreshTriggersReconnectAttempts`	`5`
Since: 4.2 Set the threshold for the `PERSISTENT_RECONNECTS` refresh trigger. Topology updates based on persistent reconnects lead only to a refresh if the reconnect process tries at least the number of specified attempts. The first reconnect attempt starts with `1`.		
Dynamic topology refresh sources	`dynamicRefreshSources`	`true`
Since: 4.2 Discover cluster nodes from topology and use the discovered nodes as the source for the cluster topology. Using dynamic refresh will query all discovered nodes for the cluster topology and calculate the number of clients for each node. If set to `false`, only the initial seed nodes will be used as sources for topology discovery and the number of clients will be obtained only for the initial seed nodes. This can be useful when using Redis Cluster with many nodes.		
Close stale connections	`closeStaleConnections`	`true`
Since: 3.3, 4.1 Stale connections are existing connections to nodes which are no longer part of the Redis Cluster. If this flag is set to `true`, then stale connections are closed upon topology refreshes. It's strongly advised to close stale connections as open connections will attempt to reconnect nodes if the node is no longer available and open connections require system resources.		
Limitation of cluster redirects	`maxRedirects`	`5`
Since: 3.1, 4.0 When the assignment of a slot-hash is moved in a Redis Cluster and a client requests a key that is located on the moved slot-hash, the Cluster node responds with a `-MOVED` response. In this case, the client follows the redirection and queries the cluster specified within the redirection. Under some circumstances, the redirection can be endless. To protect the client and also the Cluster, a limit of max redirects can be configured. Once the limit is reached, the `-MOVED` error is returned to the caller. This limit also applies for `-ASK` redirections in case a slot is set to `MIGRATING` state.		
Validate cluster node membership	`validateClusterNodeMembership`	`true`

Name	Method	Default
Since: 3.3, 4.0 Validate the cluster node membership before allowing connections to that node. The current implementation performs redirects using `MOVED` and `ASK` and allows obtaining connections to the particular cluster nodes. The validation was introduced during the development of version 3.3 to prevent security breaches and only allow connections to the known hosts of the `CLUSTER NODES` output. There are some scenarios, where the strict validation is an obstruction: `MOVED`/`ASK` redirection but the cluster topology view is stale Connecting to cluster nodes using different IP's/hostnames (e.g. private/public IP's) Connecting to non-cluster members to reconfigure those while using the RedisClusterClient connection.		

7.2.2. Request queue size and cluster

Clustered operations use multiple connections. The resulting overall-queue limit is `requestQueueSize * ((number of cluster nodes * 2) + 1)`.

7.3. SSL Connections

lettuce supports SSL connections since version 3.1 on Redis Standalone connections and since version 4.2 on Redis Cluster. Redis has no native SSL support, SSL is implemented usually by using stunnel.

An example stunnel configuration can look like:

Example 45. stunnel.conf

```
cert=/etc/ssl/cert.pem
key=/etc/ssl/key.pem
capath=/etc/ssl/cert.pem
cafile=/etc/ssl/cert.pem
delay=yes
pid=/etc/ssl/stunnel.pid
foreground = no

[redis]
accept = 127.0.0.1:6443
connect = 127.0.0.1:6479
```

Next step is connecting lettuce over SSL to Redis.

Example 46. Connecting to Redis with SSL using `RedisURI`

```
RedisURI redisUri = RedisURI.Builder.redis("localhost")
                                    .withSsl(true)
                                    .withPassword("authentication")
                                    .withDatabase(2)
                                    .build();

RedisClient client = RedisClient.create(redisUri);
```

Example 47. Connecting to Redis with SSL using String `RedisURI`

```
RedisURI redisUri = RedisURI.create("rediss://authentication@localhost/2");
RedisClient client = RedisClient.create(redisUri);
```

Example 48. Connecting to Redis Cluster with SSL using `RedisURI`

```
RedisURI redisUri = RedisURI.Builder.redis("localhost")
                                    .withSsl(true)
                                    .withPassword("authentication")
                                    .build();

RedisClusterClient client = RedisClusterClient.create(redisUri);
```

7.3.1. Limitations

lettuce supports SSL only on Redis Standalone and Redis Cluster connections. Master resolution using Redis Sentinel or Redis Master/Slave are not supported since both strategies provide Redis addresses to the native port. Redis Sentinel and Redis Master/Slave cannot provide the SSL ports.

7.3.2. Connection Procedure and Reconnect

When connecting using SSL, lettuce performs an SSL handshake before you can use the connection. Plain text connections do not perform a handshake. Errors during the handshake throw `RedisConnectionException`s.

Reconnection behavior is also different to plain text connections. If an SSL handshake fails on reconnect (because of peer/certification verification or peer does not talk SSL) reconnection will be disabled for the connection. You will also find an error log entry within your logs.

7.3.3. Certificate Chains/Root Certificate/Self-Signed Certificates

lettuce uses Java defaults for the trust store that is usually `cacerts` in your `jre/lib/security`

directory and comes with customizable SSL options via Configuring Client resources. If you need to add you own root certificate, so you can configure `SslOptions`, import it either to `cacerts` or you provide an own trust store and set the necessary system properties:

Example 49. Configuring `SslOptions` via Client options

```
SslOptions sslOptions = SslOptions.builder()
        .jdkSslProvider()
        .truststore(new File("yourtruststore.jks"), "changeit")
        .build();

ClientOptions clientOptions = ClientOptions.builder().sslOptions(sslOptions).
build();
```

Example 50. Configuring a custom trust store via System Properties

```
System.setProperty("javax.net.ssl.trustStore", "yourtruststore.jks");
System.setProperty("javax.net.ssl.trustStorePassword", "changeit");
```

7.3.4. Host/Peer Verification

By default, lettuce verifies the certificate against the validity and the common name (Name validation not supported on Java 1.6, only available on Java 1.7 and higher) of the Redis host you are connecting to. This behavior can be turned off:

```
RedisURI redisUri = ...
redisUri.setVerifyPeer(false);
```

or

```
RedisURI redisUri = RedisURI.Builder.redis(host(), sslPort())
                                .withSsl(true)
                                .withVerifyPeer(false)
                                .build();
```

7.3.5. StartTLS

If you need to issue a StartTLS before you can use SSL, set the `startTLS` property of `RedisURI` to `true`. StartTLS is disabled by default.

```
RedisURI redisUri = ...
redisUri.setStartTls(true);
```

or

```
RedisURI redisUri = RedisURI.Builder.redis(host(), sslPort())
                                    .withSsl(true)
                                    .withStartTls(true)
                                    .build();
```

7.4. Native Transports

Netty provides two platform-specific JNI transports:

- epoll on Linux
- kqueue on MacOS/BSD

Lettuce defaults to native transports if the appropriate library is available within its runtime. Using a native transport adds features specific to a particular platform, generate less garbage and generally improve performance when compared to the NIO based transport. Native transports are required to connect to Redis via Unix Domain Sockets and are suitable for TCP connections as well.

Native transports are available with:

- Linux x86_64 systems with a minimum netty version of `4.0.26.Final`, requiring `netty-transport-native-epoll`, classifier `linux-x86_64`

```
<dependency>
    <groupId>io.netty</groupId>
    <artifactId>netty-transport-native-epoll</artifactId>
    <version>${netty-version}</version>
    <classifier>linux-x86_64</classifier>
</dependency>
```

- MacOS x86_64 systems with a minimum netty version of `4.1.11.Final`, requiring `netty-transport-native-kqueue`, classifier `osx-x86_64`

```
<dependency>
    <groupId>io.netty</groupId>
    <artifactId>netty-transport-native-kqueue</artifactId>
    <version>${netty-version}</version>
    <classifier>osx-x86_64</classifier>
</dependency>
```

You can disable native transport use through system properties. Set `io.lettuce.core.epoll` respective `io.lettuce.core.kqueue` to `false` (default is `true`, if unset).

7.4.1. Limitations

Native transport support does not work with the shaded version of lettuce because of two reasons:

1. `netty-transport-native-epoll` and `netty-transport-native-kqueue` are not packaged into the shaded jar. So adding the jar to the classpath will resolve in different netty base classes (such as `io.netty.channel.EventLoopGroup` instead of `com.lambdaworks.io.netty.channel.EventLoopGroup`)
2. Support for using epoll/kqueue with shaded netty requires netty 4.1 and all parts of netty to be shaded.

See also Netty documentation on native transports.

7.5. Unix Domain Sockets

Lettuce supports since version 3.2 Unix Domain Sockets for local Redis connections.

Example 51. Connecting to Redis using RedisURI

```
RedisURI redisUri = RedisURI.Builder
                            .socket("/tmp/redis")
                            .withPassword("authentication")
                            .withDatabase(2)
                            .build();

RedisClient client = RedisClient.create(redisUri);
```

Example 52. Connecting to Redis using String RedisURI

```
RedisURI redisUri = RedisURI.create("redis-socket:///tmp/redis");
RedisClient client = RedisClient.create(redisUri);
```

Unix Domain Sockets are inter-process communication channels on POSIX compliant systems. They allow exchanging data between processes on the same host operating system. When using Redis, which is usually a network service, Unix Domain Sockets are usable only if connecting locally to a single instance. Redis Sentinel and Redis Cluster, maintain tables of remote or local nodes and act therefore as a registry. Unix Domain Sockets are not beneficial with Redis Sentinel and Redis Cluster.

Using `RedisClusterClient` with Unix Domain Sockets would connect to the local node using a socket and open TCP connections to all the other hosts. A good example is connecting locally to a standalone or a single cluster node to gain performance.

See Native Transports for more details and limitations.

7.6. Streaming API

Redis can contain a huge set of data. Collections can burst your memory, when the amount of data is too massive for your heap. Lettuce can return your collection data either as List/Set/Map or can push the data on `StreamingChannel` interfaces.

`StreamingChannel`s are similar to callback methods. Every method, which can return bulk data (except transactions/multi and some config methods) specifies beside a regular method with a collection return class also method which accepts a `StreamingChannel`. Lettuce interacts with a `StreamingChannel` as the data arrives so data can be processed while the command is running and is not yet completed.

There are 4 StreamingChannels accepting different data types:

- KeyStreamingChannel
- ValueStreamingChannel
- KeyValueStreamingChannel
- ScoredValueStreamingChannel

The result of the steaming methods is the count of keys/values/key-value pairs as `long` value.

Example 53. Streaming results for `HGETALL`

```
Long count = redis.hgetall(new KeyValueStreamingChannel<String, String>()
    {
        @Override
        public void onKeyValue(String key, String value)
        {
            ...
        }
    }, key);
```

Streaming happens real-time to the redis responses. The method call (future) completes after the last call to the StreamingChannel.

7.6.1. Examples

Example 54. ValueStreamingChannel using a Redis List

```
redis.lpush("key", "one")
redis.lpush("key", "two")
redis.lpush("key", "three")

Long count = redis.lrange(new ValueStreamingChannel<String, String>()
    {
        @Override
        public void onValue(String value)
        {
            System.out.println("Value: " + value);
        }
    }, "key",  0, -1);

System.out.println("Count: " + count);
```

will produce following output:

```
Value: one
Value: two
Value: three
Count: 3
```

7.7. Events

7.7.1. Before 3.4/4.1

lettuce can notify its users of certain events:

- Connected
- Disconnected
- Exceptions in the connection handler pipeline

You can subscribe to these events using `RedisClient#addListener()` and unsubscribe with `RedisClient.removeListener()`. Both methods accept a `RedisConnectionStateListener`.

`RedisConnectionStateListener` receives as connection the async implementation of the connection. This means if you use a sync way (e. g. `RedisConnection`) you will receive the `RedisAsyncConnectionImpl` instance

Example

```
RedisClient client = new RedisClient(host, port);
client.addListener(new RedisConnectionStateListener()
{
    @Override
    public void onRedisConnected(RedisChannelHandler<?, ?> connection)
    {

    }
    @Override
    public void onRedisDisconnected(RedisChannelHandler<?, ?> connection)
    {

    }
    @Override
    public void onRedisExceptionCaught(RedisChannelHandler<?, ?> connection, Throwable
cause)
    {

    }
});
```

7.7.2. Since 3.4/4.1

The client produces events during its operation and uses an event bus for the transport. The `EventBus` can be configured and obtained from the Client Options and is used for client- and custom events.

Following events are sent by the client:

- Connection events
- Metrics events
- Cluster topology events

Subscribing to events

The simple-most approach to subscribing to the client events is obtaining the event bus from the client's client resources.

```
RedisClient client = RedisClient.create()
EventBus eventBus = client.getresources().eventBus();

eventBus.get().subscribe(new Action1<Event>() {
    @Override
    public void call(Event event) {
        System.out.println(event);
    }
});

...
client.shutdown();
```

Calls to the `subscribe()` method will return a `Subscription`. If you plan to unsubscribe from the event stream, you can do so by calling the `Subscription.unsubscribe()` method. The event bus utilizes RxJava and the {reactive-api} to transport events from the publisher to its subscribers.

A thread of the computation thread pool (can be configured using Client Options) transports the events.

Connection events

When working with events, multiple events occur. These can be used to monitor connections or react to these. Connection events transport the local and the remote connection points. The regular order of connection events is:

1. Connected: The transport-layer connection is established (TCP or Unix Domain Socket connection established). Event type: `ConnectedEvent`
2. Connection activated: The logical connection is activated and can be used to dispatch Redis commands (SSL handshake complete, PING before activating response received). Event type: `ConnectionActivatedEvent`
3. Disconnected: The transport-layer connection is closed/reset. That event occurs on regular connection shutdowns and connection interruptions (outage). Event type: `DisconnectedEvent`
4. Connection deactivated: The logical connection is deactivated. The internal processing state is reset and the `isOpen()` flag is set to `false` That event occurs on regular connection shutdowns and connection interruptions (outage). Event type: `ConnectionDeactivatedEvent`

Metrics events

Client command metrics is published using the event bus. The current event carries command latency metrics. Latency metrics is segregated by connection or server and command which means you can get detailed statistics on every command. Connection distinction allows to see how particular connections perform. Server distinction how particular servers perform. You can configure metrics collection using Client Options.

In detail, two command latencies are recorded:

1. RTT from dispatching the command until the first command response is processed (first

response)

2. RTT from dispatching the command until the full command response is processed and at the moment the command is completed (completion)

The latency metrics provide following statistics:

- Number of commands
- min latency
- max latency
- latency percentiles

Caution, the smarty-pants details begin here

First Response Latency

The first response latency measuring begins at the moment the command sending begins (command flush on the netty event loop). That is not the time at when at which the command was issued from the client API. The latency time recording ends at the moment the client receives the first command bytes and starts to process the command response. Both conditions must be met to end the latency recording. The client could be busy with processing the previous command while the first bytes are already available to read. That scenario would be a good time to file an issue for improving the client performance. The first response latency value is good to determine the lag/network performance and can give a hint on the client and server performance.

Completion Latency

The completion latency begins at the same time as the first response latency but lasts until the time where the client is just about to call the `complete()` method to signal command completion. That means all command response bytes arrived and were decoded/processed, and the response data structures are ready for consumption for the user of the client. On completion callback duration (such as async or observable callbacks) are not part of the completion latency.

Cluster events

When using Redis Cluster, you might want to know when the cluster topology changes. As soon as the cluster client discovers the cluster topology change, a `ClusterTopologyChangedEvent` event is published to the event bus. The time at which the event is published is not necessarily the time the topology change occurred. That is because the client polls the topology from the cluster.

The cluster topology changed event carries the topology view before and after the change.

Make sure, you enabled cluster topology refresh in the Client options.

7.8. Pipelining and command flushing

Redis is a TCP server using the client-server model and what is called a Request/Response protocol. This means that usually a request is accomplished with the following steps:

- The client sends a query to the server and reads from the socket, usually in a blocking way, for

the server response.

- The server processes the command and sends the response back to the client.

A request/response server can be implemented so that it is able to process new requests even if the client did not already read the old responses. This way it is possible to send multiple commands to the server without waiting for the replies at all, and finally read the replies in a single step.

Using the synchronous API, in general, the program flow is blocked until the response is accomplished. The underlying connection is busy with sending the request and receiving its response. Blocking, in this case, applies only from a current Thread perspective, not from a global perspective.

To understand why using a synchronous API does not block on a global level we need to understand what this means. Lettuce is a non-blocking and asynchronous client. It provides a synchronous API to achieve a blocking behavior on a per-Thread basis to create await (synchronize) a command response. Blocking does not affect other Threads per se. Lettuce is designed to operate in a pipelining way. Multiple threads can share one connection. While one Thread may process one command, the other Thread can send a new command. As soon as the first request returns, the first Thread's program flow continues, while the second request is processed by Redis and comes back at a certain point in time.

Lettuce is built on top of netty decouple reading from writing and to provide thread-safe connections. The result is, that reading and writing can be handled by different threads and commands are written and read independent of each other but in sequence. You can find more details about message ordering in the Wiki. The transport and command execution layer does not block the processing until a command is written, processed and while its response is read. lettuce sends commands at the moment they are invoked.

A good example is the async API. Every invocation on the async API returns a `Future` (response handle) after the command is written to the netty pipeline. A write to the pipeline does not mean, the command is written to the underlying transport. Multiple commands can be written without awaiting the response. Invocations to the API (sync, async and starting with `4.0` also reactive API) can be performed by multiple threads.

Sharing a connection between threads is possible but keep in mind:

The longer commands need for processing, the longer other invoker wait for their results

You should not use transactional commands (`MULTI`) on shared connection. If you use Redis-blocking commands (e. g. `BLPOP`) all invocations of the shared connection will be blocked until the blocking command returns which impacts the performance of other threads. Blocking commands can be a reason to use multiple connections.

7.8.1. Command flushing

The normal operation mode of lettuce is to flush every command which means, that every command is written to the transport after it was issued. Any regular user desires this behavior. You can control command flushing since Version `3.3`.

Why would you want to do this? A flush is an expensive system call and impacts performance.

Batching, disabling auto-flushing, can be used under certain conditions and is recommended if:

- You perform multiple calls to Redis and you're not depending immediately on the result of the call
- You're bulk-importing

Controlling the flush behavior is only available on the async API. The sync API emulates blocking calls and as soon as you invoke a command, you're no longer able to interact with the connection until the blocking call ends.

The `AutoFlushCommands` state is set per connection and therefore affects all threads using the shared connection. If you want to omit this effect, use dedicated connections. The `AutoFlushCommands` state cannot be set on pooled connections by the lettuce connection pooling.

Example 55. Asynchronous Pipelining

```
StatefulRedisConnection<String, String> connection = client.connect();
RedisAsyncCommands<String, String> commands = connection.async();

// disable auto-flushing
commands.setAutoFlushCommands(false);

// perform a series of independent calls
List<RedisFuture<?>> futures = Lists.newArrayList();
for (int i = 0; i < iterations; i++) {
    futures.add(commands.set("key-" + i, "value-" + i);
}

// write all commands to the transport layer
commands.flushCommands();

// synchronization example: Wait until all futures complete
boolean result = LettuceFutures.awaitAll(5, TimeUnit.SECONDS,
                  futures.toArray(new RedisFuture[futures.size()]));

// later
connection.close();
```

Performance impact

Commands invoked in the default flush-after-write mode perform in an order of about 100Kops/sec (async/multithreaded execution). Grouping multiple commands in a batch (size depends on your environment, but batches between 50 and 1000 work nice during performance tests) can increase the throughput up to a factor of 5x.

Pipelining within the Redis docs: http://redis.io/topics/pipelining

7.9. Connection Pooling

Lettuce connections are designed to be thread-safe so one connection can be shared amongst multiple threads and Lettuce connections auto-reconnection by default. While connection pooling is not necessary in most cases it can be helpful in certain use cases. Lettuce provides generic connection pooling support.

7.9.1. Is connection pooling necessary?

Lettuce is thread-safe by design which is sufficient for most cases. All Redis user operations are executed single-threaded. Using multiple connections does not impact the performance of an application in a positive way. The use of blocking operations usually goes hand in hand with worker threads that get their dedicated connection. The use of Redis Transactions is the typical use case for dynamic connection pooling as the number of threads requiring a dedicated connection tends to be dynamic. That said, the requirement for dynamic connection pooling is limited. Connection pooling always comes with a cost of complexity and maintenance.

7.9.2. Execution Models

Lettuce supports two execution models for pooling:

- Synchronous/Blocking via Apache Commons Pool 2
- Asynchronous/Non-Blocking via a Lettuce-specific pool implementation (since version 5.1)

7.9.3. Synchronous Connection Pooling

Using imperative programming models, synchronous connection pooling is the right choice as it carries out all operations on the thread that is used to execute the code.

Prerequisites

Lettuce requires Apache's common-pool2 dependency (at least 2.2) to provide connection pooling. Make sure to include that dependency on your classpath. Otherwise, you won't be able using connection pooling.

If using Maven, add the following dependency to your `pom.xml`:

```
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-pool2</artifactId>
    <version>2.4.3</version>
</dependency>
```

Connection pool support

Lettuce provides generic connection pool support. It requires a connection `Supplier` that is used to create connections of any supported type (Redis Standalone, Pub/Sub, Sentinel, Master/Slave, Redis Cluster). `ConnectionPoolSupport` will create a `GenericObjectPool` or `SoftReferenceObjectPool`,

depending on your needs. The pool can allocate either wrapped or direct connections.

- Wrapped instances will return the connection back to the pool when called `StatefulConnection.close()`.
- Regular connections need to be returned to the pool with `GenericObjectPool.returnObject(…)`.

Basic usage

```
RedisClient client = RedisClient.create(RedisURI.create(host, port));

GenericObjectPool<StatefulRedisConnection<String, String>> pool =
ConnectionPoolSupport
               .createGenericObjectPool(() -> client.connect(), new
GenericObjectPoolConfig());

// executing work
try (StatefulRedisConnection<String, String> connection = pool.borrowObject()) {

    RedisCommands<String, String> commands = connection.sync();
    commands.multi();
    commands.set("key", "value");
    commands.set("key2", "value2");
    commands.exec();
}

// terminating
pool.close();
client.shutdown();
```

Cluster usage

```
RedisClusterClient clusterClient = RedisClusterClient.create(RedisURI.create(host,
port));

GenericObjectPool<StatefulRedisClusterConnection<String, String>> pool =
ConnectionPoolSupport
                .createGenericObjectPool(() -> clusterClient.connect(), new
GenericObjectPoolConfig());

// execute work
try (StatefulRedisClusterConnection<String, String> connection = pool.borrowObject())
{
    connection.sync().set("key", "value");
    connection.sync().blpop(10, "list");
}

// terminating
pool.close();
clusterClient.shutdown();
```

7.9.4. Asynchronous Connection Pooling

Asynchronous/non-blocking programming models require a non-blocking API to obtain Redis connections. A blocking connection pool can easily lead to a state that blocks the event loop and prevents your application from progress in processing.

Lettuce comes with an asynchronous, non-blocking pool implementation to be used with Lettuces asynchronous connection methods. It does not require additional dependencies.

Asynchronous Connection pool support

Lettuce provides asynchronous connection pool support. It requires a connection `Supplier` that is used to asynchronously connect to any supported type (Redis Standalone, Pub/Sub, Sentinel, Master/Slave, Redis Cluster). `AsyncConnectionPoolSupport` will create a `BoundedAsyncPool`. The pool can allocate either wrapped or direct connections.

- Wrapped instances will return the connection back to the pool when called `StatefulConnection.closeAsync()`.
- Regular connections need to be returned to the pool with `AsyncPool.release(…)`.

Basic usage

```
RedisClient client = RedisClient.create();

AsyncPool<StatefulRedisConnection<String, String>> pool = AsyncConnectionPoolSupport
.createBoundedObjectPool(
        () -> client.connectAsync(StringCodec.UTF8, RedisURI.create(host, port)),
BoundedPoolConfig.create());

// execute work
CompletableFuture<TransactionResult> transactionResult = pool.acquire().thenCompose
(connection -> {

    RedisAsyncCommands<String, String> async = connection.async();

    async.multi();
    async.set("key", "value");
    async.set("key2", "value2");
    return async.exec().whenComplete((s, throwable) -> pool.release(c));
});

// terminating
pool.closeAsync();

// after pool completion
client.shutdownAsync();
```

Cluster usage

```
RedisClusterClient clusterClient = RedisClusterClient.create(RedisURI.create(host,
port));

AsyncPool<StatefulRedisConnection<String, String>> pool = AsyncConnectionPoolSupport
.createBoundedObjectPool(
        () -> clusterClient.connectAsync(StringCodec.UTF8), BoundedPoolConfig.create(
));

// execute work
CompletableFuture<String> setResult = pool.acquire().thenCompose(connection -> {

    RedisAsyncCommands<String, String> async = connection.async();

    async.set("key", "value");
    return async.async.set("key2", "value2").whenComplete((s, throwable) -> pool
.release(c));
});

// terminating
pool.closeAsync();

// after pool completion
client.shutdownAsync();
```

7.10. Custom commands

Lettuce covers nearly all Redis commands. Redis development is an ongoing process and the Redis Module system is intended to introduce new commands which are not part of the Redis Core. This requirement introduces the need to invoke custom commands or use custom outputs. Custom commands can be dispatched on the one hand using Lua and the `eval()` command, on the other side lettuce 4.x allows you to trigger own commands. That API is used by lettuce itself to dispatch commands and requires some knowledge of how commands are constructed and dispatched within lettuce.

Lettuce provides two levels of command dispatching:

1. Using the synchronous, asynchronous or reactive API wrappers which invoke commands according to their nature
2. Using the bare connection to influence the command nature and synchronization (advanced)

Example using `dispatch()` on the synchronous API

```
RedisCodec<String, String> codec = StringCodec.UTF8;
RedisCommands<String, String> commands = ...

String response = redis.dispatch(CommandType.SET, new StatusOutput<>(codec),
                new CommandArgs<>(codec)
                        .addKey(key)
                        .addValue(value));
```

Example using `dispatch()` on the asynchronous API

```
RedisCodec<String, String> codec = StringCodec.UTF8;
RedisAsyncCommands<String, String> commands = ...

RedisFuture<String> response = redis.dispatch(CommandType.SET, new StatusOutput<>
(codec),
                                    new CommandArgs<>(codec)
                                        .addKey(key)
                                        .addValue(value));
```

Example using `dispatch()` on the reactive API

```
RedisCodec<String, String> codec = StringCodec.UTF8;
RedisReactiveCommands<String, String> commands = ...

Observable<String> response = redis.dispatch(CommandType.SET, new StatusOutput<>(
codec),
                                    new CommandArgs<>(codec)
                                        .addKey(key)
                                        .addValue(value));
```

Example using a `RedisFuture` command wrapper

```
StatefulRedisConnection<String, String> connection = redis.getStatefulConnection();

RedisCommand<String, String, String> command = new Command<>(CommandType.PING,
                                        new StatusOutput<>(new Utf8StringCodec()));

AsyncCommand<String, String, String> async = new AsyncCommand<>(command);
connection.dispatch(async);

// async instanceof CompletableFuture == true
```

7.10.1. Mechanics of lettuce commands

Lettuce uses the command pattern to implement to execute commands. Every time a command is invoked, lettuce creates a command object (`Command` or types implementing `RedisCommand`).

Commands can carry arguments (`CommandArgs`) and an output (subclasses of `CommandOutput`). Both are optional. The two mandatory properties are the command type (see `CommandType` or a type implementing `ProtocolKeyword`) and a `RedisCodec`. If you dispatch commands by yourself, do not reuse command instances to dispatch commands more than once. Commands that were executed once have the completed flag set and cannot be reused.

Arguments

`CommandArgs` is a container command arguments that follow the command keyword (`CommandType`). A `PING` or `QUIT` command do not require commands whereas the `GET` or `SET` commands require arguments in the form of keys and values.

The `PING` command

```
RedisCommand<String, String, String> command = new Command<>(CommandType.PING,
                                        new StatusOutput<>(StringCodec.UTF8));
```

The `SET` command

```
StringCodec codec = StringCodec.UTF8;
RedisCommand<String, String, String> command = new Command<>(CommandType.SET,
                new StatusOutput<>(codec), new CommandArgs<>(codec)
                                                .addKey("key")
                                                .addValue("value"));
```

`CommandArgs` allow to add one or more:

- key and arrays of keys
- value and arrays of values
- `String`, `long` (the Redis integer), `double`
- byte array
- `CommandType`, `CommandKeyword` and generic `ProtocolKeyword`

The sequence of args and keywords is not validated by lettuce beyond the supported data types, meaning Redis will report errors if the command syntax is not correct.

Outputs

Commands producing an output are required to consume the output. lettuce supports type-safe conversion of the response into the appropriate result types. The output handlers derive from the `CommandOutput` base class. lettuce provides a wide range of output types (see the `com.lambdaworks.redis.output` package for details). Command outputs are mostly used to return the result as the whole object. The response is available as soon as the whole command output is processed. There are cases, where you might want to stream the response instead of allocating a significant amount of memory and return the whole response as one. These types are called streaming outputs. Following implementations ship with lettuce:

- KeyStreamingOutput
- KeyValueScanStreamingOutput
- KeyValueStreamingOutput
- ScoredValueStreamingOutput
- ValueScanStreamingOutput
- ValueStreamingOutput

Those outputs take a streaming channel (see ValueStreamingChannel) and invoke the callback method (e.g. onValue(V value)) for every data element.

Implementing an own output is, in general, a good idea when you want to support a different data type, or you want to work with different types than the basic collection, map, String, and primitive types. You might get an impression of the custom types idea by taking a look on GeoWithinListOutput, which takes a bunch of strings and nested lists to construct a list of GeoWithin instances.

Please note that using an output that does not fit the command output can jam the response processing and lead to not usable connections. Use either ArrayOutput or NestedMultiOutput when in doubt, so you receive a list of objects (nested lists).

Output for the PING command

```
Command<String, String, String> command = new Command<>(CommandType.PING,
                                        new StatusOutput<>(StringCodec.UTF8));
```

Output for the HGETALL command

```
StringCodec codec = StringCodec.UTF8;
Command<String, String, Map<String, String>> command = new Command<>(CommandType
.HGETALL,
                                    new MapOutput<>(codec),
                                    new CommandArgs<>(codec).addKey(key));
```

Output for the HKEYS command

```
StringCodec codec = StringCodec.UTF8;
Command<String, String, List<String>> command = new Command<>(CommandType.HKEYS,
                                    new KeyListOutput<>(codec),
                                    new CommandArgs<>(codec).addKey(key));
```

7.10.2. Synchronous, asynchronous and reactive

Great, that you made it up to here. You might want to know now, how to synchronize the command completion, work with Futures or how about the reactive API. The simple way is using the dispatch(…) method of the according wrapper. If this is not sufficient, then continue on reading.

The `dispatch()` method on a stateful Redis connection is not opinionated at all how you are using lettuce, whether it is synchronous or reactive. The only thing this method does is dispatching the command. The response handler handles decoding the command and completing the command once it's done. The asynchronous command processing is the only operating mode of lettuce.

The `RedisCommand` interface provides methods to `complete()`, `cancel()` and `completeExceptionally()` the command. The `complete()` methods are called by the response handler as soon as the command is completed. Redis commands can be wrapped and augmented by that way. Wrapping is used when using transactions (`MULTI`) or Redis Cluster.

You are free to implement your command type or use one of the provided commands:

- Command (default implementation)
- AsyncCommand (the `CompleteableFuture` wrapper for `RedisCommand`)
- CommandWrapper (generic wrapper)
- TransactionalCommand (wraps `RedisCommand`s when `MULTI` is active)

Fire & Forget

Fire&Forget is the simple-most way to dispatch commands. You just trigger it and then you do not care what happens with it, whether it completes or not, and you don't have access to the command output:

```
StatefulRedisConnection<String, String> connection = redis.getStatefulConnection();

RedisCommand<String, String, String> command = new Command<>(CommandType.PING,
                                        new StatusOutput<>(StringCodec.UTF8));

connection.dispatch(command);
```

Asynchronous

The asynchronous API works in general with the `AsyncCommand` wrapper that extends `CompleteableFuture`. `AsyncCommand` can be synchronized by `await()` or `get()` which corresponds with the asynchronous pull style. By using the methods from the `CompletionStage` interface (such as `handle()` or `thenAccept()`) the response handler will trigger the functions ("listeners") on command completion. Lear more about asynchronous usage in the Asynchronous API topic.

```
StatefulRedisConnection<String, String> connection = redis.getStatefulConnection();

RedisCommand<String, String, String> command = new Command<>(CommandType.PING,
                                        new StatusOutput<>(StringCodec.UTF8));

AsyncCommand<String, String, String> async = new AsyncCommand<>(command);
connection.dispatch(async);

// async instanceof CompletableFuture == true
```

Synchronous

The synchronous API of lettuce uses future synchronization to provide a synchronous view.

Reactive

Reactive commands are dispatched at the moment of subscription (see Reactive API for more details on reactive APIs). In the context of lettuce this means, you need to start before calling the `dispatch()` method. The reactive API uses internally an `ObservableCommand`, but that is internal stuff. If you want to dispatch commands the ractive way, you'll need to wrap commands (or better: command supplier to be able to retry commands) with the `ReactiveCommandDispatcher`. The dispatcher implements the `OnSubscribe` API to create an `Observable<T>`, handles command dispatching at the time of subscription and can dissolve collection types to particular elements. An instance of `ReactiveCommandDispatcher` allows to create multiple `Observable`s as long as you use a `Supplier<RedisCommand>`. Commands that were executed once have the completed flag set and cannot be reused.

```
StatefulRedisConnection<String, String> connection = redis.getStatefulConnection();

RedisCommand<String, String, String> command = new Command<>(CommandType.PING,
                new StatusOutput<>(StringCodec.UTF8));
ReactiveCommandDispatcher<String, String, String> dispatcher = new
ReactiveCommandDispatcher<>(command,
                connection, false);

Observable<String> observable = Observable.create(dispatcher);
String result = observable.toBlocking().first();

result == "PONG"
```

7.11. Command execution reliability

Lettuce is a thread-safe and scalable Redis client that allows multiple independent connections to Redis.

7.11.1. General

lettuce provides two levels of consistency; these are the rules for Redis command sends:

Depending on the chosen consistency level:

- **at-most-once execution**, i. e. no guaranteed execution
- **at-least-once execution**, i. e. guaranteed execution (with some exceptions)

Always:

- command ordering in the order of invocations

7.11.2. What does *at-most-once* mean?

When it comes to describing the semantics of an execution mechanism, there are three basic categories:

- **at-most-once** execution means that for each command handed to the mechanism, that command is execution zero or one time; in more casual terms it means that commands may be lost.
- **at-least-once** execution means that for each command handed to the mechanism potentially multiple attempts are made at execution it, such that at least one succeeds; again, in more casual terms this means that commands may be duplicated but not lost.
- **exactly-once** execution means that for each command handed to the mechanism exactly one execution is made; the command can neither be lost nor duplicated.

The first one is the cheapest - the highest performance, least implementation overhead - because it can be done without tracking whether the command was sent or got lost within the transport mechanism. The second one requires retries to counter transport losses, which means keeping the state at the sending end and having an acknowledgment mechanism at the receiving end. The third is most expensive—and has consequently worst performance—because also to the second it requires a state to be kept at the receiving end to filter out duplicate executions.

7.11.3. Why No Guaranteed Delivery?

At the core of the problem lies the question what exactly this guarantee shall mean:

1. The command is sent out on the network?
2. The command is received by the other host?
3. The command is processed by Redis?
4. The command response is sent by the other host?
5. The command response is received by the network?
6. The command response is processed successfully?

Each one of these have different challenges and costs, and it is obvious that there are conditions under which any command sending library would be unable to comply. Think for example about how a network partition would affect point three, or even what it would mean to decide upon the “successfully” part of point six.

The only meaningful way for a client to know whether an interaction was successful is by receiving a business-level acknowledgment command, which is not something lettuce could make up on its own.

lettuce allows two levels of consistency; each one has its costs and benefits, and therefore it does not try to lie and emulate a leaky abstraction.

7.11.4. Message Ordering

The rule more specifically is that commands sent are not be executed out-of-order.

The following illustrates the guarantee:

- Thread `T1` sends commands `C1`, `C2`, `C3` to Redis
- Thread `T2` sends commands `C4`, `C5`, `C6` to Redis

This means that:

- If `C1` is executed, it must be executed before `C2` and `C3`.
- If `C2` is executed, it must be executed before `C3`.
- If `C4` is executed, it must be executed before `C5` and `C6`.
- If `C5` is executed, it must be executed before `C6`.
- Redis executes commands from `T1` interleaved with commands from `T2`.
- If there is no guaranteed delivery, any of the commands may be dropped, i.e. not arrive at Redis.

7.11.5. Failures and *at-least-once* execution

lettuce's *at-least-once* execution is scoped to the lifecycle of a logical connection. Redis commands are not persisted to be executed after a JVM or client restart. All Redis command state is held in memory. A retry mechanism re-executes commands that are not successfully completed if a network failure occurs. In more casual terms, when Redis is available again, the retry mechanism fires all queued commands. Commands that are issued as long as the failure persists are buffered.

at-least-once execution ensures a higher consistency level than *at-least-once* but comes with some caveats:

- Commands can be executed more than once
- Higher usage of resources since commands are buffered and sent again after reconnect

Exceptions to *at-least-once*

lettuce does not loose commands while sending them. A command execution can, however, fail for the same reasons as a normal method call can on the JVM:

- `StackOverflowError`
- `OutOfMemoryError`
- other `Error`s

Also, executions can fail in specific ways:

- The command runs into a timeout
- The command cannot be encoded
- The command cannot be decoded, because:
- The output is not compatible with the command output
- Exceptions occur while command decoding/processing. This may happen a `StreamingChannel` results in an error, or a consumer of Pub/Sub events fails while listener notification.

While the first is clearly a matter of configuration, the second deserves some thought: The command execution does not get feedback if there was a timeout. This is in general not distinguishable from a lost message. By using the Sync API, commands that exceeded their timeout are canceled. This behavior cannot be changed. When using the Async API, users can decide, how to proceed with the command, whether the command should be canceled.

Commands which run into `Exception`s while encoding or decoding reach a non-recoverable state. Commands that cannot be *encoded* are **not** executed but get canceled. Commands that cannot be *decoded* were already executed; only the result is not available. These errors are caused mostly due to a wrong implementation. The result of a command, which cannot be *decoded* is that the command gets canceled, and the causing `Exception` is available in the result. The command is cleared from the response queue, and the connection stays useable.

In general, when `Errors` occur while operating on a connection, you should close the connection and use a new one. Connections, that experienced such severe failures get into a unrecoverable state, and no further response processing is possible.

Executing commands more than once

In terms of consistency, Redis commands can be grouped into two categories:

- Idempotent commands
- Non-idempotent commands

Idempotent commands are commands that lead to the same state if they are executed more than once. Read commands are a good example for idempotency since they do not change the state of data. Another set of idempotent commands are commands that write a whole data structure/entry at once such as `SET`, `DEL` or `CLIENT SETNAME`. Those commands change the data to the desired state. Subsequent executions of the same command leave the data in the same state.

Non-idempotent commands change the state with every execution. This means, if you execute a command twice, each resulting state is different in comparison to the previous. Examples for non-idempotent Redis commands are such as `LPUSH`, `PUBLISH` or `INCR`.

Note: When using master-slave replication, different rules apply to *at-least-once* consistency. Replication between Redis nodes works asynchronously. A command can be processed successfully from lettuce's client perspective, but the result is not necessarily replicated to the slave yet. If a failover occurs at that moment, a slave takes over, and the not yet replicated data is lost. Replication behavior is Redis-specific. Further documentation about failover and consistency from Redis perspective is available within the Redis docs: http://redis.io/topics/replication

7.11.6. Switching between *at-least-once* and *at-most-once* operations

lettuce's consistency levels are bound to retries on reconnects and the connection state. By default, lettuce operates in the *at-least-once* mode. Auto-reconnect is enabled and as soon as the connection is re-established, queued commands are re-sent for execution. While a connection failure persists, issued commands are buffered.

To change into *at-most-once* consistency level, disable auto-reconnect mode. Connections cannot be longer reconnected and thus no retries are issued. Not successfully commands are canceled. New

commands are rejected.

7.11.7. Clustered operations

lettuce sticks in clustered operations to the same rules as for standalone operations but with one exception:

Command execution on master nodes, which is rejected by a `MOVED` response are tried to re-execute with the appropriate connection. `MOVED` errors occur on master nodes when a slot's responsibility is moved from one cluster node to another node. Afterwards *at-least-once* and *at-most-once* rules apply.

When the cluster topology changes, generally spoken, the cluster slots or master/slave state is reconfigured, following rules apply:

- **at-most-once** If the connection is disconnected, queued commands are canceled and buffered commands, which were not sent, are executed by using the new cluster view
- **at-least-once** If the connection is disconnected, queued and buffered commands, which were not sent, are executed by using the new cluster view
- If the connection is not disconnected, queued commands are finished and buffered commands, which were not sent, are executed by using the new cluster view

Chapter 8. Integration and Extension

8.1. Codecs

Codecs are a pluggable mechanism for transcoding keys and values between your application and Redis. The default codec supports UTF-8 encoded String keys and values.

Each connection may have its codec passed to the extended `RedisClient.connect` methods:

```
StatefulRedisConnection<K, V> connect(RedisCodec<K, V> codec)
StatefulRedisPubSubConnection<K, V> connectPubSub(RedisCodec<K, V> codec)
```

lettuce ships with predefined codecs:

- `com.lambdaworks.redis.codec.ByteArrayCodec` - use `byte[]` for keys and values
- `com.lambdaworks.redis.codec.StringCodec` - use Strings for keys and values. Using the default charset or a specified `Charset` with improved support for `US_ASCII` and `UTF-8`.
- `com.lambdaworks.redis.codec.Utf8StringCodec` - use Strings for keys and values and convert Strings using UTF-8 to store them within Redis

Publish/Subscribe connections use channel names and patterns for keys; messages are treated as values.

Keys and values can be encoded independently from each other which means the key can be a `java.lang.String` while the value is a `byte[]`. Many other constellations are possible like:

- Representing your data as JSON if your data is mapped to a particular Java type. Different types are hard to map since the codec applies to all operations.
- Serialize your data using the Java Serializer (`ObjectInputStream`/`ObjectOutputStream`). Allows type-safe conversions but is less interoperable with other languages
- Serializing your data using Kryo for improved type-safe serialization.
- Any specialized codecs like the `BitStringCodec` (see below)

8.1.1. Why `ByteBuffer` instead of `byte[]`

The `RedisCodec` interface accepts and returns `ByteBuffer`s for data interchange. A `ByteBuffer` is not opinionated about the source of the underlying bytes. The `byte[]` interface of lettuce 3.x required the user to provide an array with the exact data for interchange. So if you have an array where you want to use only a subset, you're required to create a new instance of a byte array and copy the data. The same applies if you have a different byte source (e.g. netty's `ByteBuf` or an NIO `ByteBuffer`). The `ByteBuffer`s for decoding are pointers to the underlying data. `ByteBuffer`s for encoding data can be either pure pointers or allocated memory. lettuce does not free any memory (such as pooled buffers).

8.1.2. Diversity in Codecs

As in every other segment of technology, there is no one-fits-it-all solution when it comes to Codecs. Redis data structures provide a variety of The key and value limitation of codecs is intentionally and a balance amongst convenience and simplicity. The Redis API allows much more variance in encoding and decoding particular data elements. A good example is Redis hashes. A hash is identified by its key but stores another key/value pairs. The keys of the key-value pairs could be encoded using a different approach than the key of the hash. Another different approach might be to use different encodings between lists and sets. Using a base codec (such as UTF-8 or byte array) and performing an own conversion on top of the base codec is often the better idea.

8.1.3. Multi-Threading

A key point in Codecs is that Codecs are shared resources and can be used by multiple threads. Your Codec needs to be thread-safe (by shared-nothing, pooling or synchronization). Every logical lettuce connection uses its codec instance. Codec instances are shared as soon as multiple threads are issuing commands or if you use Redis Cluster.

8.1.4. Compression

Compression can be a good idea when storing larger chunks of data within Redis. Any textual data structures (such as JSON or XML) are suited for compression. Compression is handled at Codec-level which means you do not have to change your application to apply compression. The `CompressionCodec` provides basic and transparent compression for values using either GZIP or Deflate compression:

Example 56. Compression Codec usage

```
StatefulRedisConnection<String, Object> connection = client.connect(
                CompressionCodec.valueCompressor(new SerializedObjectCodec(),
CompressionCodec.CompressionType.GZIP)).sync();

StatefulRedisConnection<String, String> connection = client.connect(
                CompressionCodec.valueCompressor(new Utf8StringCodec(),
CompressionCodec.CompressionType.DEFLATE)).sync();
```

Compression can be used with any codec, the compressor just wraps the inner `RedisCodec` and compresses/decompresses the data that is interchanged. You can build your own compressor the same way as you can provide own codecs.

8.1.5. Examples

Example 57. BitString codec

```
public class BitStringCodec implements Utf8StringCodec {
    @Override
    public String decodeValue(ByteBuffer bytes) {
        StringBuilder bits = new StringBuilder(bytes.remaining() * 8);
        while (bytes.remaining() > 0) {
            byte b = bytes.get();
            for (int i = 0; i < 8; i++) {
                bits.append(Integer.valueOf(b >>> i & 1));
            }
        }
        return bits.toString();
    }
}

StatefulRedisConnection<String, String> connection = client.connect(new
BitStringCodec());
RedisCommands<String, String> redis = connection.sync();

redis.setbit(key, 0, 1);
redis.setbit(key, 1, 1);
redis.setbit(key, 2, 0);
redis.setbit(key, 3, 0);
redis.setbit(key, 4, 0);
redis.setbit(key, 5, 1);

redis.get(key) == "00100011"
```

Example 58. JDK Serializer

```
public class SerializedObjectCodec implements RedisCodec<String, Object> {
    private Charset charset = Charset.forName("UTF-8");

    @Override
    public String decodeKey(ByteBuffer bytes) {
        return charset.decode(bytes).toString();
    }

    @Override
    public Object decodeValue(ByteBuffer bytes) {
        try {
            byte[] array = new byte[bytes.remaining()];
            bytes.get(array);
            ObjectInputStream is = new ObjectInputStream(new ByteArrayInputStream
(array));
            return is.readObject();
        } catch (Exception e) {
            return null;
        }
    }

    @Override
    public ByteBuffer encodeKey(String key) {
        return charset.encode(key);
    }

    @Override
    public ByteBuffer encodeValue(Object value) {
        try {
            ByteArrayOutputStream bytes = new ByteArrayOutputStream();
            ObjectOutputStream os = new ObjectOutputStream(bytes);
            os.writeObject(value);
            return ByteBuffer.wrap(bytes.toByteArray());
        } catch (IOException e) {
            return null;
        }
    }
}
```

8.2. CDI Support

CDI support for Lettuce is available for `RedisClient` and `RedisClusterClient`. You need to provide a `RedisURI` in order to get lettuce injected.

8.2.1. RedisURI producer

Implement a simple producer (either field producer or producer method) of `RedisURI`:

```
@Produces
public RedisURI redisURI() {
    return RedisURI.Builder.redis("localhost").build();
}
```

lettuce also supports qualified `RedisURI`'s:

```
@Produces
@PersonDB
public RedisURI redisURI() {
    return RedisURI.Builder.redis("localhost").build();
}
```

8.2.2. Injection

After declaring your `RedisURI`'s you can start using lettuce in your classes:

```
public class InjectedClient {

    @Inject
    private RedisClient redisClient;

    @Inject
    private RedisClusterClient redisClusterClient;

    @Inject
    @PersonDB
    private RedisClient redisClient;

    private RedisConnection<String, String> connection;

    @PostConstruct
    public void postConstruct() {
        connection = redisClient.connect();
    }

    public void pingRedis() {
        connection.ping();
    }

    @PreDestroy
    public void preDestroy() {
        if (connection != null) {
            connection.close();
        }
    }
}
```

8.2.3. Activating Lettuce's CDI extension

By default, you just drop lettuce on your classpath and declare at least one `RedisURI` bean. That's all.

The CDI extension registers one bean pair (`RedisClient` and `RedisClusterClient`) per discovered `RedisURI`. This means, if you do not declare any `RedisURI` producers, the CDI extension won't be activated at all. This way you can use lettuce in CDI-capable containers without even activating the CDI extension.

All produced beans (`RedisClient` and `RedisClusterClient`) remain active as long as your application is running since the beans are `@ApplicationScoped`.

8.3. Spring Support

Use Lettuce with Spring to manage the `RedisClient` and the `RedisClusterClient`. You need to specify a `RedisURI` or a URI string in order to create the client.

You can integrate with Lettuce either by using Spring Data Redis (recommended) or standalone by

providing Java/XML configuration.

8.3.1. Spring Data Redis

Using Lettuce through Spring Data Redis provides familiar Spring abstractions for Redis usage. It integrates well with other Spring components such as Spring Session or Spring Boot. See Spring Data Redis reference documentation for lettuce usage.

8.3.2. Redis Client

Lettuce Standalone/Sentinel/Pub/Sub and Master/Slave can be used through `RedisClient`. You can provide bean definitions to manage Lettuce resources inside a Spring context. Bean management can take care of resource allocation and clean up through Spring's bean lifecycle management. Using managed beans gives you the possibility to access lettuce's `RedisClient`/connections from various places inside your code. Your code can also benefit from dependency injection.

Java Configuration

Configuring lettuce using Spring's Java Configuration with `@Bean` definitions requires you to provide bean definitions. Notice the specification of `destroyMethod` to clean up resources once the application context is shut down.

```
@Configuration
public class LettuceConfig {

    @Bean(destroyMethod = "shutdown")
    ClientResources clientResources() {
        return DefaultClientResources.create();
    }

    @Bean(destroyMethod = "shutdown")
    RedisClient redisClient(ClientResources clientResources) {

        return RedisClient.create(clientResources, RedisURI.create(TestSettings.host(
), TestSettings.port()));
    }

    @Bean(destroyMethod = "close")
    StatefulRedisConnection<String, String> connection(RedisClient redisClient) {
        return redisClient.connect();
    }
}
```

XML Configuration

Use `RedisClientFactoryBean` to create a managed instance of `RedisClient` using XML-based configuration.

```
<bean id="redisClient" class="com.lambdaworks.redis.support.RedisClientFactoryBean">
    <property name="password" value="mypassword"/>
    <!-- Redis URI Format: redis://host[:port]/database -->
    <!-- Redis URI: Specify Database as Path -->
    <property name="uri" value="redis://localhost/12"/>

    <!-- Redis Sentinel URI Format: redis-
sentinel://host[:port][,host[:port][,host[:port]]/database#masterId -->
    <!-- Redis Sentinel URI: You can specify multiple sentinels. Specify Database as
Path, Master Id as Fragment. -->
    <property name="uri" value="redis-
sentinel://localhost,localhost2,localhost3/1#myMaster"/>
</bean>
```

8.3.3. Redis Cluster Client

Lettuce Redis Cluster support can be used through `RedisClusterClient`. You can provide bean definitions to manage Lettuce resources inside a Spring context. Bean management can take care of resource allocation and clean up through Spring's bean lifecycle management. Using managed beans gives you the possibility to access Lettuce's `RedisClusterClient`/connections from various places inside your code. Your code can also benefit from dependency injection.

Java Configuration

Configuring Lettuce using Spring's Java Configuration with `@Bean` definitions requires you to provide bean definitions. Notice the specification of `destroyMethod` to clean up resources once the application context is shut down.

```
@Configuration
public class LettuceConfig {

    @Bean(destroyMethod = "shutdown")
    ClientResources clientResources() {
        return DefaultClientResources.create();
    }

    @Bean(destroyMethod = "shutdown")
    RedisClusterClient redisClusterClient(ClientResources clientResources) {

        RedisURI redisURI = RedisURI.create(TestSettings.host(), 7379);

        return RedisClusterClient.create(clientResources, redisURI);
    }

    @Bean(destroyMethod = "close")
    StatefulRedisClusterConnection<String, String> clusterConnection
(RedisClusterClient clusterClient) {
        return clusterClient.connect();
    }
}
```

XML Configuration

Use `RedisClusterClientFactoryBean` to create a managed instance of `RedisClusterClient` using XML-based configuration.

```
<bean id="redisClient" class=
"com.lambdaworks.redis.support.RedisClusterClientFactoryBean">
    <property name="password" value="mypassword"/>
    <!-- Redis URI Format: redis://host[:port]/database -->
    <!-- Redis URI: Specify Database as Path -->
    <property name="uri" value="redis://localhost/12"/>
</bean>
```