<https://university.redis.com/courses/course-v1:redislabs+RU102N+2023_01/course/>

1. Intro to StackeExchange.Redis

Getting Started with StackExchange.Redis

Ok, so let's get started with StackExchange.Redis. In this bit, we'll look at some of the fundamental architectural choices of StackExchange.Redis. We'll discuss how to add it to your project, and then we'll go over how to connect to Redis from StackExchange.Redis and send the most basic of Redis Commands: [**PING**](https://redis.io/commands/ping).

# Connection Multiplexing

The most fundamental architectural feature of StackExchange.Redis is Connection Multiplexing. The library leans heavily on a class called the **ConnectionMultiplexer**. This class is responsible for arbitrating all connections to Redis, and routing all commands you want to send through the library through a single connection.

That's right, a single connection. The **ConnectionMultiplexer** opens exactly 2 connections per Redis Server, one of which is the interactive command connection to Redis, the other being the subscription connection for the pub/sub API which we'll explore later.

Like everything else in computing and software development, this approach has both benefits and tradeoffs.

## **Benefits**

The **ConnectionMultiplexer** has proven to be extremely performant and robust. It's a powerhouse for pushing commands through to Redis.

* The single connection multiplexer matches the cardinality of Redis Threads. There's only one command thread in Redis, so sending additional commands concurrently doesn't help much as they will be waiting to be serviced by the command thread.
* It minimizes the number of sockets your application needs to open and maintain and wards off Socket Exhaustion.
* The multiplexer will maximize usage of your sockets, and automatically pipeline commands sent concurrently.

## **Tradeoffs**

As with every other endeavour in software development, any architectural choice as profound as the **ConnectionMultiplexer** has some tradeoffs. We'll explore these in more detail throughout the course, but here are they are at a top level.

* Head-of-line blockages can occur with large payloads blocking out other requests.
* Blocking commands, e.g. blocking stream reads, the blocking list/sorted set commands cannot be used. This is because blocking the interactive connection will block out threads trying to use the connection concurrently.
* Transactions work a bit differently, we'll talk about them later in the course, but they are a tad different in that they don't fully support watches, and no command in a transaction is dispatched to Redis until execution time.

# Add StackExchange.Redis

 Bookmark this page

There are of course a variety of ways to Add StackExchange.Redis to your project. From the base directory of the github project, first checkout the **starting-point** tag, this will leave you with a skeleton of the projects we are going to build in this course. Move to the **src/section\_1/section1.2** directory with **cd src/section\_1/section1.2**.

This directory contains **section1.2.csproj**, which is a more or less empty csproj file. To Add StackExchange.Redis to this project, follow any of the methods outlined in [NuGet](https://www.nuget.org/packages/StackExchange.Redis), or just run: **dotnet add package StackExchange.Redis**

# 1.2 Connect to Redis!

Now that we've added StackExchange.Redis to the project, we can move into our first Hands-On exercise, connecting to Redis! You can follow along with this Hands-On by opening /src/section\_1/section1.2/section1.2.csproj in your IDE.

There are two methods that you can use to connect to Redis: **ConnectionMultiplexer.Connect** and **ConnectionMultiplexer.ConnectAsync** - either will work. You'll need the parameters that you collected at setup:

* hostname
* port
* password (if on Redis Cloud or otherwise password protected)
* username (if not **default**)

There are essentially two overloads, one taking a connection string, and the other taking an instance of the **ConfigurationOptions** class, which is definitely easier to organize, but the connection string option is often easier for a simple connection.

## **Connection String**

The Connection string is just a comma delimited string of different configuration paramaters, any hostname:port formatted argument in the connection string is treated as it's own endpoint, while any param=val formatted argument is treated as a parameter value. So the connection string:

**redis-1:6379,password=foobar**

Equates to StackExchange.Redis connecting to the host "redis-1", port 6379, and using the password "foobar" as it's password.

## **ConfigurationOptions**

The ConfigurationOptions class allows for parameterized construction of a collection of options for the multiplexer. The equivalent options to our connection string above would be:

**var options = new ConfigurationOptions**

**{**

**EndPoints = new EndPointCollection{"redis-1:6379"},**

**Password = "foobar"**

**};**

## **Ping Redis**

After you've established a connection to Redis. Let's send our very first interactive command to Redis!

We'll start off by sending the simplest of Redis commands, **PING**. First, you'll need to get a handle to an **IDatabase**, the **IDatabase** is the main interactive command interface for Redis Commands. You can grab it from the Multiplexer by calling the **GetDatabase** method on the **ConnectionMultiplexer**.

With that done, all you need to do to ping Redis is call **Ping**, print the results, and you're done!

Connect to Redis Solution

The following is what my code looked like after the previous exercise, yours might be slightly different, and that's ok! but fundamentally, you should have:

* Connected to the **ConnectionMultiplexer**
* Grabbed an **IDatabase** from the Multiplexer
* Called **Ping** on the **IDatabase**
* Printed the results from that ping.

// Start Programming Challenge

using StackExchange.Redis;

Console.WriteLine("Hello Redis!");

var muxer = ConnectionMultiplexer.Connect(new ConfigurationOptions

{

EndPoints = new EndPointCollection{"localhost:6379"}

});

var db = muxer.GetDatabase();

var res = db.Ping();

Console.WriteLine($"The ping took: {res.TotalMilliseconds} ms");

//End Programming Challenge

# 1.3 Interfaces of StackExchange.Redis

The public API of StackExchange.Redis is broken up across several critical interfaces. We'll briefly go over each of them in this section. You've actually already touched two of them **IConnectionMultiplexer** and **IDatabase**.

## **The Interfaces**

* **IConnectionMultiplexer**
* **IDatabase**
* **IServer**
* **ISubscriber**
* **ITransaction**

## **IConnectionMultiplexer**

The **IConnectionMultiplexer** is responsible for maintaining all of the connections to Redis. As I described in the previous section. It routes all the commands to Redis through a single connection for interactive commands, and a separate connection for subscription, which we'll discuss more in depth later.

The **IConnectionMultiplexer** is responsible for exposing a simple interface to get other critical interfaces of the library. Including the **IDatabase**, **ISubscriber**, and **IServer**.

## **IDatabase**

The **IDatabase** can be thought of as the primary interactive interface to Redis. It provides a single interface for your entire Redis Instance, and is the preferred interface when you are executing single commands that manipulate your application's data to Redis.

The **IDatabase**, unlike the **IServer**, abstracts the particulars of your Redis deployments away. Consequentially, if you are running in a cluster and are preforming a write, the **IDatabase** does not require you to know which server in particular you need to write to. Also, if you have many replicas per master shard in your [Cluster](https://redis.io/docs/management/scaling/) or [Sentinel](https://redis.io/docs/management/sentinel/) Redis deployments, the **IDatabase** will leverage the **ConnectionMultiplexer** to automatically distribute your reads across your deployment.

## **The IServer**

The **IServer** is an abstraction to a single instance of a Redis Server. You can grab an instance of an **IServer** by using the **IConnectionMultiplexer.GetServer** command, passing in the exact endpoint information you want to retrieve.

**IServer** has a fundamentally different role than **IDatabase** as you're going to use it to handle the server level commands. That means that in general, data modeling commands are not appropriate to be used on a server. Rather operations like checking the server's info (the basic info of Redis), it's configuration, updating it's configuration, checking memory statistics, and the like are **IServer** operations. Even scanning for the keys of a Redis server should be done at the server level.

## **ISubscriber**

The **ISubscriber** is the interface responsible for maintaining subscriptions to Redis in the pub/sub interface. Unlike the other interfaces we've looked at thus far, the subscriber does not leverage the interactive connection.

The Multiplexer explicitly opens a separate connection for subscriptions because when you subscribe to any channel on a client in Redis, the client connection converts to subscription mode. This limits the connection to only use commands that implement subscriber functionality.

True to it's name however, the Multiplexer continues to maintain a single connection per server, and all subscriptions are handled on that single connection.

You you can get an instance of an **ISubscriber** by calling the **IConnectionMultiplexer.GetSubscriber()** method.

## **ITransaction**

The **ITransaction** provides an interface for Redis [Transactions](https://redis.io/docs/manual/transactions/). Transactions in Redis differ from transactions in other databases, for a full description of transactions check out the transaction section of [RU101](https://university.redis.com/courses/ru101/).

The **ITransaction** interface is fundamentally an async interface. It exposes a very similar command set to the **IDatabase**, but it will only expose async versions of each command. That is because each command in **ITransaction** is async, as they will not be competed until after **Execute** is called. Only after the **Execute** is called can the underpinning tasks for the Transaction be awaited.

You can get an instance of an **ITransaction** by calling **IDatabase.GetTransaction()** on your **IDatabase** object.

1.4 Advanced Connections

In this section we'll explore connecting the Multiplexer to instances of Redis that are beyond the unencrypted single server variety. We'll explore:

* Connecting to instances of Redis with TLS enabled.
* Connecting to Redis Cluster Deployments.
* Connecting to Redis Sentinel Deployments

# Connect to Redis with TLS

An important feature of Redis Cloud, Redis Enterprise, and more recently with Redis 6, is Transportation Layer Security (TLS). To learn more about how to use use TLS with Redis, see [RU330: Redis Security](https://university.redis.com/courses/ru330/) or the TLS article on [redis.io](https://redis.io/docs/manual/security/encryption/).

## **.NET Framework vs .NET Core**

When it comes to connecting to Redis with TLS it's important that we differentiate between .NET Framework and .NET Core. Because of the difference in the **[SSLStream](https://learn.microsoft.com/en-us/dotnet/api/system.net.security.sslstream" \t "_blank)** API in Framework vs Core, Core is substantially more configurable.

### Configuring TLS in the .NET Framework

To configure TLS in the .NET Framework you have two options. You can either:

* Set up the certificate selection and validation event handlers.
* Set up certificate/password environment variables.

#### Event Handlers

There are two event handlers that you can use with the **ConfigurationOptions** object to configure TLS. Neither of these can be used from a connection string so you must use the **ConfigurationOptions** if you need to use them.

The first of these is the **ConfigurationOptions.CertificateSelection** event handler. This event handler selects the certificate that you will use to authenticate with the Redis Server.

**configurationOptions.CertificateSelection += delegate**

**{**

**var cert = new X509Certificate2(PATH\_TO\_CERT\_FILE, "");**

**return cert;**

**};**

The other event handler, **ConfigurationOptions.CertificateValidation**, is used to check the validity of the Redis Server's certificates:

**configurationOptions.CertificateValidation += (sender, certificate, chain, errors) =>**

**{**

**var isValid = true;**

**// insert check certificate logic here**

**return isValid;**

**};**

#### Environment Variables

There are also two environment variables that the **ConnectionMultiplexer** will check if there is no certificate selection delegate set. This is useful if you are using a derivative library of StackExchange.Redis that does not provide direct access to the **ConfigurationOptions** object.

* **SERedis\_ClientCertPfxPath**: provides the multiplexer the path to the certificate file.
* **SERedis\_ClientCertPassword**: Is the password for the certificate file if applicable.

### Connecting in .NET Core

In .NET Core (3.1, 5, 6, 7+) **[SslClientAuthenticationOptions](https://learn.microsoft.com/en-us/dotnet/api/system.net.security.sslclientauthenticationoptions" \t "_blank)** was added as an optional way of configuring an **SslStream**. This provides a great deal of extra flexibility when it comes to configuring TLS settings Redis.

Using the **SslClientAuthenticationOptions** delegate in **ConfigurationOptions** you can configure:

* Allowed SSL/TLS protocols
* TLS/SSL Cipher Suites allowed in the cipher negotiation
* Certificate selection delegate
* Certificate validation delegate

Example:

**options.SslClientAuthenticationOptions = new Func<string, SslClientAuthenticationOptions>(**

**hostName => new SslClientAuthenticationOptions**

**{**

**EnabledSslProtocols = SslProtocols.Tls12 | SslProtocols.Tls13**

**});**

# Connecting to a Redis Cluster

Of the "advanced" Redis Connections, cluster is probably the easiest to understand. Connecting to an OSS Cluster instance is the same as connecting to a standalone Redis Instance. The only exception is that when you are listing endpoints, you'll want to use more than one. This is so that if the endpoint you are trying to reach has failed over, the Multiplexer still has a chance to connect to the other master instances.

An example **ConfigurationOptions** for a cluster instance might look something along the lines of this, notice how the Endpoints collection takes multiple endpoints which the **ConnectionMultiplexer** can use as backups if one of the endpoints should fail to respond (possibly because it's failed over).

**var options = new ConfigurationOptions**

**{**

**// add and update parameters as needed**

**EndPoints = new EndPointCollection{"redis-1:6379", "redis-2:6379", "redis-3:6379"}**

**};**

# Connect to Redis Sentinel

Connecting to [Redis Sentinel](https://redis.io/docs/management/sentinel/) is a bit different than connecting to other instances of Redis.

The key difference is that rather than connecting to the master server, you connect to one of the 'sentinels' - the instances of Redis responsible for monitoring your master and replicas, detecting fail-overs, and promoting new masters. Additionally, you must specify the **ServiceName**, which corresponds to the master name that you tell the sentinels to monitor when configuring them.

**var options = new ConfigurationOptions**

**{**

**EndPoints = new EndPointCollection{"sentinel-1:26379"},**

**ServiceName = "sentinel"**

**};**

**var muxer = ConnectionMultiplexer.Connect(options);**

# 1.5 Pipelining

Pipelining is a critically important concept for maximizing throughput to Redis. When you need to execute multiple commands against Redis, and the intermediate results can be temporarily ignored, pipelining can drastically reduce the number of round trips required to Redis, which can drastically increase performance, as many operations are hundreds of times faster than the Round Trip Time (RTT).

With StackExchange.Redis, there are two ways to pipeline commands, either implicitly with the Async API, and explicitly with the **IBatch** API.

## **Implicit Pipelining with Async API**

If you use the async version of a command, the command will be automatically pipelined to Redis. If you use **await** to await the result of a task dispatched by one of these async commands, it will wait until the command is complete before returning control. However, if you group a set of tasks dispatched by the async methods together and await them all in one shot, the **ConnectionMultiplexer** will automatically find an efficient way to pipeline the commands to Redis, so that you can cut down the number of round trips significantly.

## **Explicit Pipelining with IBatch**

You can also be much more explicit about pipelining commands. The **IBatch** API only provides the async interface for commands. You can set up however many commands you want to pipeline with those async methods, preserving the tasks as they will provide you the results.

When you have all the commands that you want pipelined dispatched, you can call Execute to run them all. This will pipeline all of your commands in one contiguous block to Redis. Using this method, no other commands will be interleaved with your batched commands from the client. However, if there are other clients sending commands to Redis, it's possible that their commands will be interleaved with the batched commands.

# Hands-On Pipelining

As we just reviewed previously, pipelining can be crucial for increasing throughput on your Redis instance. In this exercise, we are going to look at running the same collection of tasks in three modes.

* Serially and un-pipelined
* Implicitly pipelined
* Explicitly pipelined with IBatch

The completed source is in the course's [repository](https://github.com/redislabs-training/ru102n) under **/src/section\_1/section1.5/section1.5.csproj**. We'll be using the **ping** command in this case, but this will extrapolate to any command, and any sequence of commands where the intermediate results are not needed. If you'd like to start from the beginning you can checkout the **starting-point** branch and open the csproj file finding the **// TODO for Coding Challenge Start here on starting-point branch** in **Program.cs**.

## **Preamble**

Before we run our examples, let's initialize a Multiplexer, get an instance of an **IDatabase** and create and start a **StopWatch**:

**using System.Diagnostics;**

**using StackExchange.Redis;**

**var options = new ConfigurationOptions**

**{**

**EndPoints = new EndPointCollection { "localhost:6379" }**

**};**

**var muxer = ConnectionMultiplexer.Connect(options);**

**var db = muxer.GetDatabase();**

**var stopwatch = Stopwatch.StartNew();**

## **Un-Pipelined**

Sending a serial chain of commands that are not pipelined is pretty straight forward. For the sake of consistency with the other examples, we'll use the async version of the **Ping** Command. We'll just call and await the result 1000 times. This will cause the Multiplexer to wait for the command to complete between runs, and prevent it from pipelining the commands automatically.

**// un-pipelined commands incur the added cost of an extra round trip**

**for (var i = 0; i < 1000; i++)**

**{**

**await db.PingAsync();**

**}**

**Console.WriteLine($"1000 un-pipelined commands took: {stopwatch.ElapsedMilliseconds}ms to execute");**

## **Implicitly Pipelined**

Now let's try using implicit pipelining, in this instance, we will not await each task as it's dispatched. Rather we will collect all the tasks and await them all en masse at the end. Each Task is responsible for containing the results of the command after the command completes:

**// If we run out async tasks to StackExchange.Redis concurrently, the library**

**// will automatically manage pipelining of these commands to Redis, making**

**// them significantly more performant as we remove most of the round trips to Redis.**

**var pingTasks = new List<Task<TimeSpan>>();**

**// restart stopwatch**

**stopwatch.Restart();**

**for (var i = 0; i < 1000; i++)**

**{**

**pingTasks.Add(db.PingAsync());**

**}**

**await Task.WhenAll(pingTasks);**

**Console.WriteLine($"1000 automatically pipelined tasks took: {stopwatch.ElapsedMilliseconds}ms to execute, first result: {pingTasks[0].Result}");**

## **Explicit Pipelining with IBatch**

Finally, let's explicitly pipeline all of our pings using an **IBatch**. An **IBatch** will guarantee that the client sends the entire batch to Redis in one shot, with no other commands interleaved in the pipeline. This is slightly different behavior than our implicit pipelining as in the case of implicit pipelining, commands may be interleaved with any other commands the client was executing at the time.

To explicitly pipeline these commands we'll follow a similar pattern, in this case, however, we will use the **IDatabase.CreateBatch()** method to create the batch, and use the batch's async methods to 'dispatch' the the tasks. It's important to note here that unlike in our implicit case, the tasks will not be truly dispatched until after the **IBatch.Execute()** method is called, if you try awaiting any of the tasks before then, you can accidentally deadlock your command. After calling **Execute**, you can then await all of the tasks.

**// clear our ping tasks list.**

**pingTasks.Clear();**

**// Batches allow you to more intentionally group together the commands that you want to send to Redis.**

**// If you employee a batch, all commands in the batch will be sent to Redis in one contiguous block, with no**

**// other commands from the client interleaved. Of course, if there are other clients to Redis, commands from those**

**// other clients may be interleaved with your batched commands.**

**var batch = db.CreateBatch();**

**// restart stopwatch**

**stopwatch.Restart();**

**for (var i = 0; i < 1000; i++)**

**{**

**pingTasks.Add(batch.PingAsync());**

**}**

**batch.Execute();**

**await Task.WhenAll(pingTasks);**

**Console.WriteLine($"1000 batched commands took: {stopwatch.ElapsedMilliseconds}ms to execute, first result: {pingTasks[0].Result}");**

# [Section 2 - Using Redis Data Structures with .NET](https://university.redis.com/courses/course-v1:redislabs+RU102N+2023_01/courseware/2504017c40ce422eae155215c9a30553/4e99c538aed14719beab9772e901ec89/?child=first#section-2-using-redis-data-structures-with-net-child)

# 2.1 String Operations

Redis Strings are the simplest of the Redis Data Structures. They map a single Redis Key to a single Redis Value. In spite of their simplicity they have a variety of capabilities that make them useful across a number of use cases.

# Hands-On with Strings

In this Hands-On we'll be exploring how to use Redis Strings from StackExchange.Redis. You can either create your own project to follow along with and connect a multiplexer, or you can open **/src/section\_2/section2.1/section2.1.csproj** in your IDE.

## **Simple Sets and Gets**

The simplest things you can do with a string are to set it and get it, to do this all you need to do is use the **StringSet** and **StringGet** methods:

**var instructorNameKey = new RedisKey("instructors:1:name");**

**db.StringSet(instructorNameKey, "Steve");**

**var instructor1Name = db.StringGet(instructorNameKey);**

**Console.WriteLine($"Instructor 1's name is: {instructor1Name}");**

Now, if you want to append a string onto an existing string value, you can use the **StringAppend** method:

**db.StringAppend(instructorNameKey, " Lorello");**

**instructor1Name = db.StringGet(instructorNameKey);**

**Console.WriteLine($"Instructor 1's full name is: {instructor1Name}");**

## **String Numerics**

There are alternative encodings of strings beyond a raw character string. If a string is set to a numeric value, you can use the **StringIncrement** methods on it.

There are two overloads of the **StringIncrement** method, if you pass in a long, it will increment the key by a long value:

**var tempKey = "temperature";**

**db.StringSet(tempKey, 42);**

**var tempAsLong = db.StringIncrement(tempKey, 5);**

**Console.WriteLine($"New temperature: {tempAsLong}");**

You can use that same overload, leaving the increment empty, and it will default to using an integer increment of 1.

**tempAsLong = db.StringIncrement(tempKey);**

**Console.WriteLine($"New Temp: {tempAsLong}");**

Now, if you need to increment by a floating point, you can pass in a double as the increment like so:

**var tempAsDouble = db.StringIncrement(tempKey, .5);**

**Console.WriteLine($"New temperature: {tempAsDouble}");**

Note that if you increment by a floating point, you will not be able to increment by an integer anymore, as you've changed the encoding of the string.

## **String Options**

There are a couple of important options to take note of when you are setting strings.

### Expiration

You can set an expiration on a string when you set it by using the expiry option. The expiry is a **TimeSpan**, there are a variety of ways you can create them. In this instance we'll just create a super short lived key by using the **FromSeconds** method to initialize a key that will live for 1 second.

**db.StringSet("temporaryKey", "hello world", expiry: TimeSpan.FromSeconds(1));**

### Conditional Set

You can also make your **StringSet** conditional using the when option. The when option uses the **When** enum, and has three possible values.

* **Always**: This is the default, always set the key.
* **Exists**: Set only if the key exists
* **NotExists**: Set only if the key does not exist.

Using them is as simple as passing in the **when** option:

**var conditionalKey = "ConditionalKey";**

**var conditionalKeyText = "this has been set";**

**// You can also specify a condition for when you want to set a key**

**// For example, if you only want to set a key when it does not exist**

**// you can by specifying the NotExists condition**

**var wasSet = db.StringSet(conditionalKey, conditionalKeyText, when: When.NotExists);**

**Console.WriteLine($"Key set: {wasSet}");**

**// Of course, after the key has been set, if you try to set the key again**

**// it will not work, and you will get false back from StringSet**

**wasSet = db.StringSet(conditionalKey, "this text doesn't matter since it won't be set", when:When.NotExists);**

**Console.WriteLine($"Key set: {wasSet}");**

**// You can also use When.Exists, to set the key only if the key already exists**

**wasSet = db.StringSet(conditionalKey, "we reset the key!");**

**Console.WriteLine($"Key set: {wasSet}");**

# 2.2 Redis Lists

Redis Lists are doubly linked lists that allow you to push and pop from the front and tail. This allows you to build what are in essence queues and stacks of Redis Strings. The StackExchange.Redis library provides an intuitive interface for working with Lists in the **IDatabase** that we will go over in this section.

# Hands-On with Redis Lists

In this Hands-On we'll be exploring how to use Redis Lists from StackExchange.Redis. You can either create your own project to follow along with and connect a multiplexer, or you can open **/src/section\_2/section2.2/section2.2.csproj** in your IDE.

We'll be exploring lists using two lists of strings, fruits and vegetables. So, now that we have our multiplexer initialized and we have a handle to the **IDatabase**, our first task is to clean out any prior list, so we start each program run with a clean slate.

**var fruitKey = "fruits";**

**var vegetableKey = "vegetables";**

**db.KeyDelete(new RedisKey[] { fruitKey, vegetableKey });**

## **Pushing to Lists**

Lists are doubly linked lists, meaning you can push or pop from either side of the list. However, any sort of indexed access becomes an O(N) operation where N is the the number of elements you need to traverse to reach the index. Let's start out by pushing.

### Push Left

Let's push some fruit to the left side of the list. Each time we push a fruit to the left, we are adding that fruit to the head of the list.

**db.ListLeftPush(fruitKey, new RedisValue[]{"Banana", "Mango", "Apple", "Pepper", "Kiwi", "Grape"});**

Now let's try printing out the first element in this list. Lists are indexed left to right, consequentially the last fruit that we pushed left, will be the 0th index "Grape":

**Console.WriteLine($"The first fruit in the list is: {db.ListGetByIndex(fruitKey, 0)}");**

And if we accessed the tail of the list, with the index -1, we'd get the first element we pushed into the left side of the list, "Banana":

**Console.WriteLine($"The last fruit in the list is: {db.ListGetByIndex(fruitKey, -1)}");**

#### Push Right

In our vegetable list, we'll now push right. Since we are pushing right, the item in question is added to the tail of the list, making it the last element in the list if we iterate over the list.

**db.ListRightPush(vegetableKey, new RedisValue[]{"Potato", "Carrot", "Asparagus", "Beet", "Garlic", "Tomato"});**

Now if we try to access the first element in the vegetable list, we'll get the first element that we pushed to the right, "Potato".

**Console.WriteLine($"The first vegetable in the list is: {db.ListGetByIndex(vegetableKey, 0)}");**

Conversely if we access the last element, we'll get "Tomato" - which is not a vegetable at all!

**Console.WriteLine($"The last vegetable in the list is: {db.ListGetByIndex(vegetableKey, -1)}");**

## **Enumerate a List**

To enumerate a list, you can use the **ListRange** method. If you pass in a start index, the range will start from there, and if you pass in a stop index, it will stop there, if you pass in neither a start nor a stop, it will pull back the whole list.

**Console.WriteLine($"Fruit indexes 0 to -1: {string.Join(", ", db.ListRange(fruitKey))}");**

**Console.WriteLine($"Vegetables index 0 to -2: {string.Join(", ", db.ListRange(vegetableKey, 0, -2))}");**

## **Move elements Between Lists**

You can also transfer elements between lists using the **ListMove** method This accepts two list keys, as well as the source side and destination side. These are the sides of the list you will pop from and push to respectively. Let's transfer "Tomato" from our vegetable list to our fruit list, since it's in fact a fruit and not a vegetable.

**db.ListMove(vegetableKey, fruitKey, ListSide.Right, ListSide.Left);**

## **List as a Queue**

You can user a Redis List as a defacto FIFO Queue by pushing and popping from different sides. Conventionally you'd push left, pop right:

**Console.WriteLine("Enqueuing Celery");**

**db.ListLeftPush(vegetableKey, "Celery");**

**Console.WriteLine($"Dequeued: {db.ListRightPop(vegetableKey)}");**

## **List as a Stack**

You can also get your lists to act as LIFO stacks, by pushing and popping from the same, by convention you'd typically use the left side.

**Console.WriteLine("Pushing Grapefruit");**

**db.ListLeftPush(fruitKey, "Grapefruit");**

**Console.WriteLine($"Popping Fruit: {string.Join(",", db.ListLeftPop(fruitKey, 2))}");**

## **Searching Lists**

Redis Lists also allow you to find the index of a particular item using the **ListPosition** method.

**Console.WriteLine($"Position of Mango: {db.ListPosition(fruitKey, "Mango")}");**

## **List Size**

And finally, you use the **ListLength** method to determine the size of a given list.

**Console.WriteLine($"There are {db.ListLength(fruitKey)} fruits in our Fruit List");**

# 2.3 Redis Sets

Redis Sets are an implementation of a [mathematical set](https://en.wikipedia.org/wiki/Set_(mathematics)). Like mathematical sets they have a number of key properties:

* They are unordered.
* They do not allow duplication, so there are no repeated members.
* They can be combined together to create new sets.

In this section we'll learn about how to leverage these properties using Redis Sets in .NET.

# Hands-On with Redis Sets

In this Hands-On we'll be exploring how to use Redis Sets from StackExchange.Redis. You can either create your own project to follow along with and connect a multiplexer, or you can open /src/section\_2/section2.3/section2.3.csproj in your IDE.

In this example we'll be working with 5 sets of users.

* ActiveUsers
* InactiveUsers
* OnlineUsers
* OfflineUsers
* AllUsers

Let's start out by clearing out the sets we'll be working with from Redis:

**var allUsersSet = "users";**

**var activeUsersSet = "users:state:active";**

**var inactiveUsersSet = "users:state:inactive";**

**var offlineUsersSet = "users:state:offline";**

**db.KeyDelete(new RedisKey[]{allUsersSet, activeUsersSet,inactiveUsersSet, offlineUsersSet});**

## **Populate Sub Sets**

Now let's go about populating our active, inactive, and offline user sets. To do this, we'll use the **SetAdd** Method. This method is variadic so we can do it in one command for each set.

**db.SetAdd(activeUsersSet, new RedisValue[]{"User:1","User:2"});**

**db.SetAdd(inactiveUsersSet, new RedisValue[]{"User:3", "User:4"});**

**db.SetAdd(offlineUsersSet, new RedisValue[] {"User:5", "User:6", "User:7"});**

## **Combining sets to get all users**

You'll notice that we did not populate our **allUsersSet** set. If we consider active, inactive, offline to be an exhaustive list of states, we can use the set combination operations to get a set with all of our users. We can even use **SetCombineAndStore** to store those combined users in our all users key.

**db.SetCombineAndStore(SetOperation.Union, allUsersSet, new RedisKey[]{activeUsersSet, inactiveUsersSet, offlineUsersSet});**

## **Check Membership**

Sets do not allow duplication, but they allow very rapid O(1) membership checks. So if we wanted to check to see if **User:6** is offline we could do so very easily:

**var user6Offline = db.SetContains(offlineUsersSet, "User:6");**

**Console.WriteLine($"User:6 offline: {user6Offline}");**

## **Enumerate Set**

When you want to enumerate the members of a set, you have two options: you can enumerate them all in one shot, or you can scan over the set and enumerate everything. We'll go over how to do each of those here.

### Enumerate Entire Set

If you want to guarantee that you are enumerating the entire set in one round trip, you can do so by using the **SetMembers** method. This will use the **SMEMBERS** command in Redis. If your set is relatively compact (under 1000 members), this is a perfectly valid way to pull back all of your set members.

**Console.WriteLine($"All Users In one shot: {string.Join(", ", db.SetMembers(allUsersSet))}");**

### Enumerate Set in Chunks

The alternate way to enumerate a set is to enumerate it with **SetScan**, which will create a Set Enumerator, and use the **SSCAN** command to scan over the entire set until the set is exhausted.

**Console.WriteLine($"All Users with scan : {string.Join(", ", db.SetScan(allUsersSet))}");**

## **Move Elements Between Sets**

A very normal operation you might need to perform with sets is to move elements between them. For example if **User:1** were to move offline, you can use **SetMove** to move them from the active user set to the offline user set.

**Console.WriteLine("Moving User:1 from active to offline");**

**var moved = db.SetMove(activeUsersSet, offlineUsersSet, "User:1");**

**Console.WriteLine($"Move Successful: {moved}");**

# 2.4 Sorted Sets

Sorted Sets are similar conceptually to sets, however unlike regular sets, sorted sets store and retrieve members in order using the member's score. In this section we'll learn how to use Sorted Sets using the StackExchange.Redis library. We'll learn how to add to, combine, move between, and enumerate members with sorted sets. [Read more about Sorted Sets on redis.io](https://redis.io/docs/data-types/sorted-sets/).

# Hands-On with Redis Sorted Sets in .NET

In this Hands-On we'll be exploring how to use Redis Sorted Sets from StackExchange.Redis. You can either create your own project to follow along with and connect a multiplexer, or you can open **/src/section\_2/section2.4/section2.4.csproj** in your IDE.

To illustrate the usage of Sorted Sets, we'll be using five sets. Four of them will be connected under the same domain, ***users:age***, ***users:lastAccess***, ***users:highScores***, ***users:mostRecentlyActive***, and we'll have one set ***names*** to demonstrate lexicographic ordering. First let's start by purging all these sets, that way we can run this repeatedly without any other preamble.

**var userAgeSet = "users:age";**

**var userLastAccessSet = "users:lastAccess";**

**var userHighScoreSet = "users:highScores";**

**var namesSet = "names";**

**var mostRecentlyActive = "users:mostRecentlyActive";**

**db.KeyDelete(new RedisKey[]{userAgeSet, userLastAccessSet, userHighScoreSet, namesSet, mostRecentlyActive});**

## **Populate Sets**

Now that we've cleared everything out, we can populate the sets. We'll add six members to each set. To add an item to a set you will add a **SortedSetEntry**, which is a member name and score. Because these commands are variadic, we will be passing in arrays of **SortedSetEntry**. For our purposes, we will set users with Id 1-6, for our users sets, and then for our names set, we will add an arbitrary set of names. For our user sets we'll provide scores that are relevant for the key, and importantly for the names set. Which we intend to sort lexicographically, we will provide a score of 0.

**db.SortedSetAdd(userAgeSet,**

**new SortedSetEntry[]**

**{**

**new("User:1", 20),**

**new("User:2", 23),**

**new("User:3", 18),**

**new("User:4", 35),**

**new("User:5", 55),**

**new("User:6", 62)**

**});**

**db.SortedSetAdd(userLastAccessSet,**

**new SortedSetEntry[]**

**{**

**new("User:1", 1648483867),**

**new("User:2", 1658074397),**

**new("User:3", 1659132660),**

**new("User:4", 1652082765),**

**new("User:5", 1658087415),**

**new("User:6", 1656530099)**

**});**

**db.SortedSetAdd(userHighScoreSet,**

**new SortedSetEntry[]**

**{**

**new("User:1", 10),**

**new("User:2", 55),**

**new("User:3", 36),**

**new("User:4", 25),**

**new("User:5", 21),**

**new("User:6", 44)**

**});**

**db.SortedSetAdd(namesSet,**

**new SortedSetEntry[]**

**{**

**new("John", 0),**

**new("Fred", 0),**

**new("Bob", 0),**

**new("Susan", 0),**

**new("Alice", 0),**

**new("Tom", 0)**

**});**

## **Check Score**

To check a score of a member, simply use **SortedSetScore** passing in the key name and the user id.

**var user3HighScore = db.SortedSetScore(userHighScoreSet, "User:3");**

**Console.WriteLine($"User:3 High Score: {user3HighScore}");**

## **Check Rank**

To check a user's rank, you can use the **SortedSetRank** method. With this method you can pass in the key, the User Id, and a direction that you want the sorting to be done in. In this case, when we say "Rank" we mean we want the high score coming up first. Hence we'll use the **Descending** direction.

**var user2Rank = db.SortedSetRank(userHighScoreSet, "User:2", Order.Descending);**

**Console.WriteLine($"User:2 Rank: {user2Rank}");**

## **Range Queries**

Probably the most popular way to query members from a stream is by using the "Range" methods. There are three ways that you can run ranges on a sorted set.

* By Rank
* By Score
* By Lex

There are also several other options that you can add to these queries to change the directions of these ranges, to determine whether you will get the scores along with the members, and finally to decide whether you want to store the output of the range, rather than pulling the members back in one shot.

### Range By Rank

Ranging over a sorted set by rank is the default method for ranges with sorted sets. In the case of a range by rank, there are only three things you need to know:

1. What rank do you want to start at?
2. What rank do you want to stop at?
3. What direction do you want the set to be sorted in?

With this information it's straightforward to start ranging over your sorted sets by rank, just use the **SortedSetRange** command passing in the relevant arguments. Let's try printing out the three highest scorers. To do this, we'll just go from index zero to two, and order the sorted set descending:

**var topThreeScores = db.SortedSetRangeByRank(userHighScoreSet, 0, 2, Order.Descending);**

**Console.WriteLine($"Top three: {string.Join(", ", topThreeScores)}");**

## **Range By Score**

Ranging a Sorted Set by Score is also pretty straightforward. There are two major differences:

* You are using the member's score rather than their rank within the set.
* You have to decide whether you want the scores you are using to be inclusive or exclusive.

With that decided, you can use the **SortedSetRangeByScore** method to range over your sorted set. Let's find all the users between 18 and 30 inclusively with their ages, since we want to pull back the ages as well, we will use the **SortedSetRangeByScoreWithScores** method.

**var eighteenToThirty = db.SortedSetRangeByScoreWithScores(userAgeSet, 18, 30, Exclude.None);**

**Console.WriteLine($"Users between 18 and 30: {string.Join(", ", eighteenToThirty)}");**

## **Lexicographic Ranges**

When all the scores within a sorted set are set to the same score, traditionally 0, there is a lexicographic ordering mechanic available. If you wanted to alphabetize all of the names in our names sorted set, all you would have to do is use the **SortedSetRangeByValue** method:

**var namesAlphabetized = db.SortedSetRangeByValue(namesSet);**

**Console.WriteLine($"Names Alphabetized: {string.Join(",",namesAlphabetized)}");**

If you wanted to find a proper alphabetic range, for example, all the names that start with between an "A" and "J", you can do that as well, these comparisons will be based off ASCII encoding and will be based off what **memcmp** returns. The Sorted Sets section of RU101 has a rich, detailed view as to how this works exactly. But for our purposes here, if we want to find all the names that start with between "A" and "J", we'd actually want to end with an exclusive "K" at the end like so:

**var namesBetweenAandJ = db.SortedSetRangeByValue(namesSet, "A", "K", Exclude.Stop);**

**Console.WriteLine($"Names between A and J: {string.Join(", ", namesBetweenAandJ)}");**

## **Combining Sorted Sets**

We can also combine sorted sets together to help us answer more interesting questions. Say for example we wanted to find the three most recently active players, and then determine the rank order of those three by high score. Well, we have an unpopulated as of yet sorted set ***users:mostRecentlyActive*** that we can populate using the **SortedSetRangeAndStore** method:

**db.SortedSetRangeAndStore(userLastAccessSet,mostRecentlyActive, 0, 2, order: Order.Descending);**

With that done, we can use the intersection sorted set operation to find the intersection between the most recently active players, and the player scores set. We can then weight the high score to 1, and the last access time to 0, producing only the high score. This will return the players and their scores in ascending order, but we can reverse this easily with a call to **Reverse** at the end.

**var rankOrderMostRecentlyActive = db.SortedSetCombineWithScores(SetOperation.Intersect, new RedisKey[]{userHighScoreSet, mostRecentlyActive}, new double[]{1,0}).Reverse();**

**Console.WriteLine($"Highest Scores Most Recently Active: {string.Join(", ", rankOrderMostRecentlyActive)}");**

# 2.5 Redis Hashes with StackExchange.Redis

Hashes are a simple, but very powerful data structure available within Redis. Hashes are similar to dictionaries in that they allow you to organize a set of key-value pairs at a particular key within Redis.

In this section we'll look at how to use hashes using the StackExchange.Redis library.

# Hands on with Redis Hashes

In this Hands-On, we'll be exploring how to use Redis Hashes from StackExchange.Redis. You can either create your own project to follow along with and connect a multiplexer, or you can open **/src/section\_2/section2.5/section2.5.csproj** in your IDE.

## **Collect and Clean out Keys**

In this example we'll be working with some pretty simple flat objects which we'll use hashes to organize. We'll be using three person hashes stored at the keys.

* **person:1**
* **person:2**
* **person:3**

Before we get started with our example, we want to make sure that we remove these hashes so there isn't anything stale in Redis. Let's create keys for each of them and delete them.

**var person1 = "person:1";**

**var person2 = "person:2";**

**var person3 = "person:3";**

**db.KeyDelete(new RedisKey[]{person1, person2, person3});**

## **Create the Hashes**

Next, we'll want to create our three hashes, to create a hash, use the **HashSet** method, passing in an array of **HashEntry** structs, which are simple key-value pairs.

**db.HashSet(person1, new HashEntry[]**

**{**

**new("name","Alice"),**

**new("age", 33),**

**new("email","alice@example.com")**

**});**

**db.HashSet(person2, new HashEntry[]**

**{**

**new("name","Bob"),**

**new("age", 27),**

**new("email","robert@example.com")**

**});**

**db.HashSet(person3, new HashEntry[]**

**{**

**new("name","Charlie"),**

**new("age", 50),**

**new("email","chuck@example.com")**

**});**

## **Increment a Field in a Hash**

Much like Redis Strings, you can increment fields in hashes. To do this, use the **HashIncrement** method, passing in the incrementor you want to use. If you pass nothing in, it will increment by 1.

**var newAge = db.HashIncrement(person3, "age");**

**Console.WriteLine($"person:3 new age: {newAge}");**

## **Retrieve a Field from a Hash**

Retrieving a field from a hash is simple. All you need to do is use the **HashGet** method, passing in the key and field you want to get.

**var person1Name = db.HashGet(person1, "name");**

**Console.WriteLine($"person:1 name: {person1Name}");**

## **Get All Fields From a Hash**

There are two ways to retrieve all the fields from a hash. Which one is appropriate to use is dependent on how large your hash is.

If your hash is relatively small, in Redis terms this means less than 1000 fields. You can use **HashGetAll**.

If you are working with a very large hash with many thousands of fields, you may want to use **HashScan** instead. **HashScan** allows you to paginate over your hash. This will decrease the amount of time that Redis is busy servicing any one request, but will require multiple round trips to Redis.

### HashGetAll

**var person2Fields = db.HashGetAll(person2);**

**Console.WriteLine($"person:2 fields: {string.Join(", ", person2Fields)}");**

### HashScan

**var person3Fields = db.HashScan(person3);**

**Console.WriteLine($"person:3 fields: {string.Join(", ", person3Fields)}");**

# 2.6 Redis Streams

Redis Streams are an append-only log like data structure that allows you to enqueue messages from producers to be consumed by consumers in your application. They are a powerful data structure with a rich feature set, and for a full explanation as to how and why to use Redis Streams, you can checkout [RU202: Redis Streams.](https://university.redis.com/courses/ru202/)

In this section, we'll explore using Redis Streams in .NET using the StackExchange.Redis library. We'll learn how to:

* Add Messages to a Stream.
* Read Messages from a Stream.
* Read Messages from a Stream in a Consumer Group.

## **Stream Limitations in StackExchange.Redis**

Due to the multiplexed nature of StackExchange.Redis, it's important to note at the top that there is no mechanism for using the blocking paradigms available within the stream reading operations.

This means that the Stream Read operations, **StreamRead** & **StreamReadGroup**, will not be able to use the **XREAD** and **XREADGROUP** block timer or the special ***$*** id to read only new messages.

# Hands-On with Redis Streams

In this Hands-On, we'll be exploring how to use Redis Streams from StackExchange.Redis. You can either create your own project to follow along with and connect a multiplexer, or you can open **/src/section\_2/section2.6/section2.6.csproj** in your IDE.

## **Collect and Clean Out Stream Keys**

During this exercise we'll be creating streams for two different sensors. We'll call them ***sensor:1*** and ***sensor:2***. Our first task, so that we can run this app repeatedly and predictably, will be to initialize those keys, to clear out those sensor's current stream state.

**var sensor1 = "sensor:1";**

**var sensor2 = "sensor:2";**

**db.KeyDelete(new RedisKey[]{sensor1, sensor2});**

## **Adding Items to a Stream**

Adding an item to a stream is as simple as calling **StreamAdd**.

However, for the purposes of this example we are going to do something a bit different. We are going to be dispatching all of our **StreamAdd**/**StreamRead** methods in tasks that will continually push/poll results from Redis. We'll have a **Console.ReadKey** at the end to allow you to terminate the program by pressing any key.

So, for our **StreamAdd** operation we will, in an endless loop, produce new messages every second. Now, so that there's some variation, every five seconds we'll try to increment or decrement the temperature and humidity picked up by the sensor by 1.

**var rnd = new Random();**

**Task.Run(async () =>**

**{**

**long numInserted = 0;**

**var s1Temp = 28;**

**var s2Temp = 5;**

**var s1Humid = 35;**

**var s2Humid = 87;**

**while (true)**

**{**

**await db.StreamAddAsync(sensor1, new[]**

**{**

**new NameValueEntry("temp", s1Temp),**

**new NameValueEntry("humidity", s1Humid)**

**});**

**await db.StreamAddAsync(sensor2, new[]**

**{**

**new NameValueEntry("temp", s2Temp),**

**new NameValueEntry("humidity", s2Humid)**

**});**

**await Task.Delay(1000);**

**numInserted++;**

**if (numInserted % 5 == 0)**

**{**

**s1Temp = s1Temp + rnd.Next(3) - 2;**

**s2Temp = s2Temp + rnd.Next(3) - 2;**

**s1Humid = Math.Min(s1Humid + rnd.Next(3) - 2, 100);**

**s2Humid = Math.Min(s2Humid + rnd.Next(3) - 2, 100);**

**}**

**}**

**});**

**//put all your future code above here!**

**Console.ReadKey();**

## **Read from Stream**

To read from a Redis Stream, you'll need to specify a stream position for each stream you want to read. Then, you can pass in those stream positions to the **StreamRead** method. This responds with a set of streams, each of which contains entries, if there are any messages in the stream to read from.

It's important that after each stream read, that the stream position be updated, this prevents the reader from re-reading the same message over and over again.

To accomplish all of this, add the following to just above the **Console.ReadKey()** at the bottom of the file.

**Task.Run(async () =>**

**{**

**var positions = new Dictionary**

**{**

**{ sensor1, new StreamPosition(sensor1, "0-0") },**

**{ sensor2, new StreamPosition(sensor2, "0-0") }**

**};**

**while (true)**

**{**

**var readResults = await db.StreamReadAsync(positions.Values.ToArray(), countPerStream: 1);**

**if (!readResults.Any(x => x.Entries.Any()))**

**{**

**await Task.Delay(1000);**

**continue;**

**}**

**foreach (var stream in readResults)**

**{**

**foreach (var entry in stream.Entries)**

**{**

**Console.WriteLine($"{stream.Key} - {entry.Id}: {string.Join(", ", entry.Values)}");**

**positions[stream.Key!] = new StreamPosition(stream.Key, entry.Id);**

**}**

**}**

**}**

**});**

## **Consumer Groups**

Consumer Groups in Redis Streams allow for coordinated reads of messages across consumers. These consumers can operate independently of each other without having to worry about duplicate processing of the same messages.

### Create a Consumer Group

To Create a consumer group, you just need to call **StreamCreateConsumerGroup** passing in the stream key, the group name, and the starting position. We'll create an average consumer group. This will be responsible for maintaining a running average of the telemetry gotten from each sensor.

**db.StreamCreateConsumerGroup(sensor1, "average", "0-0");**

**db.StreamCreateConsumerGroup(sensor2, "average", "0-0");**

### Read from a Consumer Group

Reading from a consumer group is slightly different than reading without a consumer group. You use the **StreamReadGroup** method, and you do not have to keep track of the ***Id*** that you want to read from.

Instead, you'll use the special ***>*** id, which tells it to respond with the next message for that consumer group. We'll be monitoring average temperatures for each of our sensors, so naturally we'll need to keep track of how many messages we've seen so far, and the cumulative temperature observed.

After we've finished processing the result of a stream message, we then need to acknowledge the message with Redis. This will allow Redis to remove the stream message from it's Pending Entries List (PEL).

**var groupName = "tempAverage";**

**db.StreamCreateConsumerGroup(sensor1, groupName, "0-0");**

**db.StreamCreateConsumerGroup(sensor2, groupName, "0-0");**

**Task.Run(async()=>**

**{**

**var tempTotals = new Dictionary { { sensor1, 0 }, { sensor2, 0 } };**

**var messageCountTotals = new Dictionary() { { sensor1, 0 }, { sensor2, 0 } };**

**var consumerName = "consumer:1";**

**var positions = new Dictionary**

**{**

**{ sensor1, new StreamPosition(sensor1, ">") },**

**{ sensor2, new StreamPosition(sensor2, ">") }**

**};**

**while (true)**

**{**

**var result = await db.StreamReadGroupAsync(positions.Values.ToArray(), groupName, consumerName, countPerStream: 1);**

**if (!result.Any(x => x.Entries.Any()))**

**{**

**await Task.Delay(1000);**

**continue;**

**}**

**foreach (var stream in result)**

**{**

**foreach (var entry in stream.Entries)**

**{**

**var temp = (int)entry.Values.First(x => x.Name == "temp").Value;**

**messageCountTotals[stream.Key!]++;**

**tempTotals[stream.Key!] += temp;**

**var avg = tempTotals[stream.Key!]/messageCountTotals[stream.Key!];**

**Console.WriteLine($"{stream.Key} average Temp = {avg:0.###}");**

**await db.StreamAcknowledgeAsync(stream.Key, groupName, entry.Id);**

**}**

**}**

**}**

**});**

## **Run the App**

At this point, all that's left to do is to run the application with **dotnet run**. This will run the app which will then spit out telemetry until you press any key.

# 2.7 Lua scripting in StackExchange.Redis

StackExchange.Redis exposes the Redis [Lua Scripting](https://redis.io/docs/manual/programmability/eval-intro/) API. This allows you to run multiple commands on the Redis server in sequence, and use the intermediate results of those commands within the context of the script.

Since Redis is single threaded, and each command within Redis executes in isolation, we can leverage Redis Scripts to handle some thorny problems where we might need to use the result of one command in another command.

Getting fully into the weeds of Lua Scripting is outside the scope of this course. There's good coverage of this topic in [RU101 Introduction to Redis Data Structures](https://university.redis.com/courses/ru101/) if you want to learn more. But for the purposes of this course. We'll simply be discussing how to work with the Lua Scripts in the context of StackExchange.Redis.

## **Prepared Scripts**

Prepared scripts are the preferred way to interact with scripting in StackExchange.Redis. Prepared scripts tend to be more readable than their unprepared compatriots, because they allow for argument substitution rather than leaning heavily on the ***ARGV*** and ***KEYS*** arrays to pass in arguments and keys to the script.

Prepared scripts are also registered with the library's internal script cache. This tracks which scripts have been registered with which servers, and handles automatic retries of scripts after a server's script cache has been cleared. In other words, the prepared script API removes a lot of the complexity from using Lua scripts in Redis. In this section we'll briefly go over how to Prepare, run, and use the results from a script.

# Hands-On with Redis Scripts

In this Hands-On, we'll be exploring how to use Prepared Lua Scripts from StackExchange.Redis. You can either create your own project to follow along with and connect a multiplexer, or you can open **/src/section\_2/section2.7/section2.7.csproj** in your IDE.

## **Preparing Scripts**

Prepared Scripts allow you to parameterize your scripts in a way that just isn't possible with traditional Lua scripts.

In a prepared Lua script, you can use the **@** symbol to denote an identifier that will be substituted when the script is called. You can then pass in objects with your named parameters. StackExchange.Redis will then differentiate what goes in through the ***KEYS*** array versus the ***ARGV*** array based on the type provided for the parameter. All ***RedisKey*** values will be treated as keys, and thus used to determine which shard to execute the script against.

Let's take a look at a simple script that will auto-increment an id based off of an id key that's stored in Redis, and use that Id to create the key for the value that we're inserting.

**var scriptText = @"**

**local id = redis.call('incr', @id\_key)**

**local key = 'key:' .. id**

**redis.call('set', key, @value)**

**return key**

**";**

**var script = LuaScript.Prepare(scriptText);**

**var key1 = db.ScriptEvaluate(script, new {id\_key=(RedisKey)"autoIncrement", value="A String Value"});**

**var key2 = db.ScriptEvaluate(script, new {id\_key=(RedisKey)"autoIncrement", value="Another String Value"});**

**Console.WriteLine($"Key 1: {key1}");**

**Console.WriteLine($"Key 2: {key2}");**

Notice how in the anonymous constructor we're using to create the parameters **id\_key** is being cast to a **RedisKey**. This will make that key go through the ***KEYS*** array, and it will use that key when determining which shard to execute against. Keep in mind that in the context of this script we are dynamically creating a key in the script. In a clustered environment, you would want to make sure that you are using hashtags to ensure that the key belongs on the shard that the script is being run on, the script has been simplified for demonstration purposes.

If you'd like to learn more about sharding and hashtags check out [RU301: Running Redis at Scale](https://university.redis.com/courses/ru301/).

# 2.8 Redis Transactions

[Transactions](https://redis.io/docs/manual/transactions/) in Redis allow you to apply multiple commands in in isolation to Redis sequentially. StackExchange.Redis exposes the **ITransaction** which operationally is similar to the **IBatch** in that it allows you to dispatch a series of tasks, which you can then await after calling the **Execute** method to run the transaction.

Unlike **IBatch** however, all commands sent through a transaction are guaranteed to be executed sequentially. And you can apply some limited conditions to the transaction to allow it to self terminate if a watched key changes. In this section we'll learn how to work with Transactions in StackExchange.Redis.

# Hands on With Redis Transactions

In this Hands-On, we'll be exploring how to use Redis Transactions from StackExchange.Redis. You can either create your own project to follow along with and connect a multiplexer, or you can open **/src/section\_2/section2.8/section2.8.csproj** in your IDE.

Redis transactions operate in a similar way to the way that Batches operate with two big caveats.

* All commands under a Redis Transaction will be sent to Redis at the same time, as opposed to traditional transactions where commands can be sent in multiple round trips.
* You cannot watch keys in StackExchange.Redis transactions. But: you can add conditions to the transaction that will prevent the transaction from moving forward in certain conditions.

## **A Simple Transaction**

Before we get into anything complex. Let's take a look at a simple transaction. Imagine we wanted to add a simple person hash to Redis. But in addition to adding the simple person hash, we also wanted to add some secondary indexes to it. Keep in mind that we will explore in later sections **SIGNIFICANTLY** more effective ways to index your data in Redis, but old-style OSS secondary indexing provides a great illustration of transactions.

Anyway, we have a person hash with three fields, name, age, and postal code and we will index all three fields. So that our data is consistent, we want to add the hash and update all the indexes at the same time. This is a perfect place for transactions as all the commands will execute in isolation.

We can do all this by initializing an **ITransaction** with **db.CreateTransaction**. We can then execute commands against that transaction, and close it out with **Execute**.

**var transaction = db.CreateTransaction();**

**transaction.HashSetAsync("person:1", new HashEntry[]**

**{**

**new ("name", "Steve"),**

**new ("age", 32),**

**new ("postal\_code", "32999")**

**});**

**transaction.SortedSetAddAsync("person:name:Steve", "person:1", 0);**

**transaction.SortedSetAddAsync("person:postal\_code:32999", "person:1", 0);**

**transaction.SortedSetAddAsync("person:age", "person:1", 32);**

**var success = transaction.Execute();**

**Console.WriteLine($"Transaction Successful: {success}");**

## **Add a Condition to a Transaction**

Redis Transactions typically allow you to watch a key to see if the key has been touched at all while the transaction was being created and fleshed out. Due to the multiplexed nature of StackExchange.Redis, this isn't something that can be done practically as multiple transactions' watches can be interleaved with each other.

However, what StackExchange.Redis does allow is for is for you to add conditions that must be satisfied in order for the transaction to complete. Let's try this out.

Say that we want our person's age to be incremented to 33, but we want to confirm that the person's age before the transaction executes is in fact what we think it is: 32.

We can add a condition to the transaction. Then, when the transaction executes, it will set the watch on Redis, query the hash, check the condition, and only if the condition passes, execute the transaction. Naturally if the state changes in the interim, the key is being watched, so the transaction will simply fail to execute once dispatched.

**transaction.AddCondition(Condition.HashEqual("person:1", "age", 32));**

**transaction.HashIncrementAsync("person:1", "age");**

**transaction.SortedSetIncrementAsync("person:age", "person:1", 1);**

**success = transaction.Execute();**

**Console.WriteLine($"Transaction Successful: {success}");**

## **Failed Transaction conditions**

There are a couple of reasons that a transaction could fail to complete. If between the point where watch was added, and the point when the transaction is transmitted to Redis, another command comes along and touches the watched key(s), then the transaction will fail to execute.

Additionally if you set a condition in the transaction, and the condition proves false, it will fail to execute. In both cases, the **transaction.Execute** will respond with false to indicate that the transaction did not succeed. Let's try getting this to fail by setting the condition to something we know to be false, e.g. the age we just incremented to 33 is 31.

**transaction.AddCondition(Condition.HashEqual("person:1", "age", 31));**

**transaction.HashIncrementAsync("person:1", "age");**

**transaction.SortedSetIncrementAsync("person:age", "person:1", 1);**

**success = transaction.Execute();**

**Console.WriteLine($"Transaction Successful: {success}");**

In this case, the transaction will simply fail, and return false when executed.

# 2.9 Redis Pub/Sub

[Pub/Sub](https://redis.io/docs/manual/pubsub/) allows for a subscriber to listen for events on channels in Redis using the Publisher/Subscriber messaging pattern.

Pub/Sub is unique in StackExchange.Redis as it is the only thing the library does that is not on the main interactive connection. Earlier in the course I mentioned that StackExchange.Redis opens two connection to Redis at a time.

The first of these is the interactive connection that you use to send all of your interactive commands to Redis on. The other is the subscriber connection. There's a really good reason for this, after subscribing to a channel in Redis, Redis flips the connection over to Subscriber mode which does not allow any non pub/sub related commands across, allowing it to send messages back to the client unhindered by other traffic on the channel.

In this section, we'll be learning how to use the pub/sub API in StackExchange.Redis. How to subscribe to channels, and handle messages that are dispatched from Redis to the client.

# Hands-On with Redis Pub/Sub

In this Hands-On, we'll be exploring how to use Redis Pub/Sub from StackExchange.Redis. You can either create your own project to follow along with and connect a multiplexer, or you can open **/src/section\_2/section2.9/section2.9.csproj** in your IDE.

## **Get an ISubscriber**

In addition to the **IDatabase** instance that we've needed in most other examples in this course, we'll also need to grab an instance of an **ISubscriber**, to get one, we'll simply need to call **ConnectionMultiplexer.GetSubscriber**. We'll have a cancellation token source so we can cut off our producer tasks when it's time, so let's also initialize that.

**var subscriber = muxer.GetSubscriber();**

**var cancellationTokenSource = new CancellationTokenSource();**

**var token = cancellationTokenSource.Token;**

## **Sequential vs Concurrent**

### Simple Sequential

When working with the subscriber, there's a decision you need to make, you can either process messages sequentially or concurrently. If you need your messages delivered in order, you're going to want to use the sequential delivery mechanism. If you do not pass a delegate to the **Subscribe** method, it produces a channel object that you can listen to messages on.

**var channel = await subscriber.SubscribeAsync("test-channel");**

**channel.OnMessage(msg =>**

**{**

**Console.WriteLine($"Sequentially received: {msg.Message} on channel: {msg.Channel}");**

**});**

## **Simple Concurrent**

To listen to messages concurrently, you need only pass a delegate into the **Subscribe** method. That delegate takes two arguments, the channel the message was sent to, and the message itself.

**await subscriber.SubscribeAsync("test-channel", (channel, value) =>**

**{**

**Console.WriteLine($"Received: {value} on channel: {channel}");**

**});**

### Create the Producer

After we've subscribed, we can set up a task to repeatedly send messages to our channel over and over again, and we'll throw a **Console.ReadKey** at the end, so that we can stop the app with any key. This will produce messages that will appear to both subscriptions, given that they are separate subscribers insofar as StackExchange.Redis is concerned. However because of the multiplexed nature of StackExchange.Redis, Redis will only see one subscriber.

**var basicSendTask = Task.Run(async () =>**

**{**

**var i = 0;**

**while (!token.IsCancellationRequested)**

**{**

**await db.PublishAsync("test-channel", i++);**

**await Task.Delay(1000);**

**}**

**});**

**// put all other producer/subscriber stuff above here.**

**Console.ReadKey();**

**// put cancellation & unsubscribe down here.**

## **Subscribe to Pattern**

The Pub/Sub API allows you to also subscribe to channel patterns, using simple glob patterns. This is basically the only difference between subscribing to patterns vs not subscribing to patterns.

If you wanted to subscribe to the prefix *pattern:*, you'd simply need to make the pattern ***pattern:\****, then whenever you send any message to a channel matching that pattern, it will be picked up. So let's also add a producer that will publish to channels matching that pattern with random GUIDs.

**var patternSendTask = Task.Run(async () =>**

**{**

**var i = 0;**

**while (!token.IsCancellationRequested)**

**{**

**await db.PublishAsync($"pattern:{Guid.NewGuid()}", i++);**

**await Task.Delay(1000);**

**}**

**});**

## **Unsubscribe**

Now that we've subscribed to our channels, the last thing to do is to unsubscribe. Let's take a look at how to do this in stages.

### Unsubscribe from Sequential Channel

We can unsubscribe from the sequential channel we created earlier by calling **Unsubscribe** on the channel object. We'll add another **Console.ReadKey** after this so that we can observe that the channel stops producing messages.

**Console.WriteLine("Unsubscribing to a single channel");**

**await channel.UnsubscribeAsync();**

**Console.ReadKey();**

### Unsubscribe Concurrent Subscriber

To unsubscribe from the subscriber that was running concurrently, we just need to call **Unsubscribe** on the subscriber, this will unsubscribe for all channels and delegates listening to that channel, so if we had done this before our sequential unsubscribe it would have been unsubscribed as well.

**Console.WriteLine("Unsubscribing whole subscriber from test-channel");**

**await subscriber.UnsubscribeAsync("test-channel");**

**Console.ReadKey();**

### Unsubscribe from Everything

If you need to unsubscribe from everything, just call **UnsubscribeAll**, and it will cut off all subscriptions to Redis.

**Console.WriteLine("Unsubscribing from all");**

**await subscriber.UnsubscribeAllAsync();**

**Console.ReadKey();**

# 3 Redis the ASP.NET Way

We'd be remiss if we went through an entire course about Redis, and did not mention ASP.NET or ASP.NET Core. In this unit we'll be looking at two of the more popular integrations of Redis in the ASP.NET ecosystem.

* Caching
* Session State Management

We'll be looking at how to do both of these in both ASP.NET and ASP.NET Core. We'll prioritize core as that's definitely the future of the ecosystem, but there are always a distinct possibility that you might be working on an older ASP.NET project that could use an integration.

# 3.1 The Derived Client Ecosystem

In the .NET Redis Ecosystem, there are a variety of clients that are derived clients of StackExchange.Redis. In this unit we'll be looking at a few of them that are specifically built around ASP.NET and ASP.NET Core.

In general the derived clients of StackExchange.Redis take many of the same parameters as StackExchange.Redis does. However there are some important exceptions which we'll explore later in this unit.

In general these clients are configured along with the rest of your application, in the case of ASP.NET Core, they are typically best configured at application startup when configuring your application services.

Both of the cases we are going to explore here are heavily integrated into the ASP.NET stack, and you'll be surprised with how simple it is to get up and running with them.

# 3.2 Caching in ASP.NET Core

Using Redis as a Cache in ASP.NET Core is as simple as using the Distributed Cache API that's available for any manner of cache that you might use in ASP.NET Core. The only change that occurs when you add Redis to your caches in ASP.NET Core is that ASP.NET Core will use Redis as its cache provider. The remainder of the operations are abstracted away from us so we don't have to think about them. That means that fundamentally, the feature set provided by the Redis Cache Provider is limited to the **IDistributedCache** API.

In this section we'll be encountering our Employee Aggregation System for the first time, where we'll use a simple model, and running some aggregations against a SQLite database with Entity Framework to see how one might use a caching framework to remove the need to repeat major computation.

# Introducing our Employee Example

For our employee aggregations example, we'll keep it simple, we'll have two components of our model, Employees and Sales. They'll look like this:

public class Employee

{

public int EmployeeId { get; set; }

public string Name { get; set; }

public List<Sale> Sales { get; } = new();

}

public class Sale

{

public int SaleId { get; set; }

public int Total { get; set; }

public int EmployeeId { get; set; }

public Employee Employee { get; set; }

}

We'll use Entity Framework and a local SQLite database for our example. If you are using a remote Redis Database, keep in mind that you'll incur some latency over the network when going to Redis vs going to our SQLite database, so if you want to truly compare apples to apples you can either run Redis locally (see the course setup instructions), or you can use a remote SQL Database by updating the adapter we're using in our example.

# Hands On with Caching in ASP.NET Core

In this Hands-On, we'll be exploring how to use Caching in ASP.NET Core. You'll likely want to open up the csproj file and checkout the *starting-point* branch before proceeding, unless of course you'd prefer to build everything from scratch :) the csproj file is located at **/src/section\_3/section3.2/section3.2.csproj** it's highly recommended that you use Visual Studio or Rider for this Hands-On.

Much of the sinew for this is already built, even if you are starting from the start point, we have a **SalesContext** that contains our model, as well as the **EntityFramework** bits to migrate the database correctly. All you need to do to migrate the database to where we need it for this example, is to from the **/src/section\_3/section3.2** directory run the following commands:

**dotnet tool install --global dotnet-ef**

**dotnet ef database update**

The first of these commands will install the dotnet-ef tool for the CLI, the second will create and migrate the Database for you.

## **Add Redis to Your Services**

After we've migrated the database, we'll need to make sure that we are adding Redis as a caching layer for our app. To do this, you'll just need to open ***Program.cs***, and find the comment ***// TODO Section 3.2 Step 1*** below that you'll just have the builder add the StackExchange Redis Cache, passing in a function to configure StackExchange.Redis.

**builder.Services.AddStackExchangeRedisCache(x => x.ConfigurationOptions = new ConfigurationOptions**

**{**

**EndPoints = { "localhost:6379" },**

**Password = ""**

**});**

Adjust the cache settings to your endpoint/password.

## **Clear Cache in Init Service**

When the application restarts, we want to reset the items that we are going to cache, as part of this, we'll need to open the ***InitService.cs***, a hosted service that runs at startup to clean up and re-seed the database, we'll need to add some logic to have it clean up the cache as well. We have a scope factory already DI'd into this service, so we can pull out the **IDistributedCache** fairly easily. Then, it's a matter of clearing out the relevant records for our app. Add The following code immediately below the **// TODO Section 3.2 Step 2** comment in ***InitService.cs***:

**var cache = scope.ServiceProvider.GetRequiredService();**

**var cachePipe = new List**

**{**

**cache.RemoveAsync("top:sales", cancellationToken),**

**cache.RemoveAsync("top:name", cancellationToken),**

**cache.RemoveAsync("totalSales", cancellationToken)**

**};**

**cachePipe.AddRange(salesDb.Employees.Select(employee => cache.RemoveAsync($"employee:{employee.EmployeeId}:avg", cancellationToken)));**

**await Task.WhenAll(cachePipe);**

## **Dependency Inject IDistributedCache**

With the cache added to our DI container, we will need to inject it into our controller. Open up ***Controllers/EmployeeController.cs*** in there, add the **IDistributedCache** as a readonly field.

**private readonly IDistributedCache \_cache;**

Then, modify the **EmployeeController** constructor to accepted an **IDistributedCache**, when you're done, it should look like this:

**public EmployeeController(SalesContext salesDb, IDistributedCache cache)**

**{**

**\_cache = cache;**

**\_salesDb = salesDb;**

**}**

## **Employee Aggregations**

As we discussed earlier, when the application boots it will randomly seed the database with thousands of records. We have some API calls here that will do some aggregating on our employee's sales figures. There's three different aggregations we'll be performing.

* The Top Salesperson and their sales
* A given sales person's average sales
* The Total Sales of all the salespeople

None of these, with the relatively small amount of data we're working with (~50,000 records), are particularly long running operations - probably on the order of 50-150 ms.

However, keep in mind that if we need access to these types of values repeatedly, we're going to be performing a lot of extra work, and of course, if we're doing anything that involves a UI, even this relatively small latency is pretty rough. That's where Redis comes in, we're going to be storing these records in our Redis Cache using the **IDistributedCache** abstraction.

### Caching Top Salesperson

Our first task here will be caching the top sales-person. This is a relatively straight forward LINQ query to go to SQLite, compute the top sales-person and their sales, and to get the information for our application, you can see it already in the **GetTopSalesperson** method.

**var topSalesperson = await \_salesDb.Employees.Select(x=>new {Employee = x, sumSales = x.Sales**

**.Sum(x=>x.Total)}).OrderByDescending(x=>x.sumSales)**

**.FirstAsync();**

Now, realistically, this will take time to compute - somewhere on the order of 50-150 ms for our smallish sample here. If we are going to run this operation repeatedly, it'd behove us to do something with this data so that we don't have to go back to our SQL database each time and recompute it.

So, we'll cache it using our **IDistributedCache**.

Find the place marked ***// TODO Section 3.2 step 3***, and add the following code to it.

**var cacheOptions = new DistributedCacheEntryOptions { AbsoluteExpirationRelativeToNow = TimeSpan.FromMinutes(5) };**

**var topSalesInsertTask = \_cache.SetStringAsync("top:sales", topSalesperson.sumSales.ToString(), cacheOptions);**

**var topNameInsertTask = \_cache.SetStringAsync("top:name", topSalesperson.Employee.Name, cacheOptions);**

**await Task.WhenAll(topSalesInsertTask, topNameInsertTask);**

This will add the total sales of the top seller and the their name to the cache, and it will invalidate that after 5 minutes. With that done, we need to add some logic up front to check the cache to see if those keys exist.

If they do, then we can short circuit the rest of this process, and return those results. Find the mark ***// TODO Section 3.2 step 4*** and add the following above it.

**var topSalesTask = \_cache.GetStringAsync("top:sales");**

**var topNameTask = \_cache.GetStringAsync("top:name");**

**await Task.WhenAll(topSalesTask, topNameTask);**

**if (!string.IsNullOrEmpty(topSalesTask.Result) && !string.IsNullOrEmpty(topNameTask.Result))**

**{**

**stopwatch.Stop();**

**return new Dictionary()**

**{**

**{ "sum\_sales", topSalesTask.Result },**

**{ "employee\_name", topNameTask.Result },**

**{ "time", stopwatch.ElapsedMilliseconds }**

**};**

**}**

## **Cache Individual Average Sale**

Next, we'll look at calculating the average sale of a given individual salesperson. To do that we'll have and endpoint that takes an Id, which we'll use to run a simple average on their sales like so:

**var avg = await \_salesDb.Employees.Include(x => x.Sales).Where(x=>x.EmployeeId == id).Select(x=>x.Sales.Average(y=>y.Total)).FirstAsync();**

Again, we'll want to cache this by setting a key in Redis, find the spot marked ***// TODO Section 3.2 step 5*** and add:

**var key = $"employee:{id}:avg";**

**var cacheResult = await \_cache.GetStringAsync(key);**

**if (cacheResult != null)**

**{**

**stopwatch.Stop();**

**return new Dictionary**

**{**

**{ "average", double.Parse(cacheResult) },**

**{ "elapsed", stopwatch.ElapsedMilliseconds }**

**};**

**}**

To create the key that we'll use. Then, underneath the mark ***// TODO Section 3.2 step 6***, add the following to set that key to the computed average. In this case, we'll set it to have a 30 minute sliding expiration, what that means is that the key will persist unless it is not touched for 30 minutes.

**await \_cache.SetStringAsync(key, avg.ToString(CultureInfo.InvariantCulture), options: new DistributedCacheEntryOptions{SlidingExpiration = TimeSpan.FromMinutes(30)});**

## **Cache Total Sales**

Finally, we'll want to cache the total sales that the whole organization created, this is probably the simplest of the calculations we're performing as there's no logic beyond summing up all the sales. However this is still a linear complexity process, so it'd still behoove us to cache it. So, let's start out by adding our cache checking logic, find the section marked ***// TODO Section 3.2 step 7*** and add the following:

**var cacheResult = await \_cache.GetStringAsync("totalSales");**

**if (cacheResult != null)**

**{**

**stopwatch.Stop();**

**return new Dictionary()**

**{**

**{ "Total Sales", long.Parse(cacheResult) },**

**{ "elapsed", stopwatch.ElapsedMilliseconds }**

**};**

**}**

This might be a metric that you run every day, so let's have it expire at midnight tomorrow morning, below mark ***// TODO Section 3.2 step 8***, add the following:

**await \_cache.SetStringAsync("totalSales", totalSales.ToString(CultureInfo.InvariantCulture), new DistributedCacheEntryOptions(){AbsoluteExpiration = DateTime.Today.AddDays(1)});**

## **Run the App**

All That's left to do now is run the app. To do this, all you need to do that is run **dotnet run** from the console in the same project as the ***.csproj*** file.

Then you can navigate to <http://localhost:5053/swagger/index.html> in your browser and you can play with the API.

You'll notice that each of these methods is pretty snappy regardless, but after the first try for each, you'll see a latency drop down to about a millisecond on each call (if you are running Redis locally).

# 3.3 Caching in ASP.NET

Before we get started here, just a quick note... the sections on ASP.NET are completely optional, they are here because there are many developers still using ASP.NET, so this can be quite useful to them.

But the reality is that ASP.NET is rapidly becoming a largely legacy framework with the emergence of ASP.NET Core. So if you don't think the sections on ASP.NET will be useful to you, feel free to skip them.

None of the questions on the exam will involve this material.

Doing simple caching in ASP.NET is the only one of the use cases we'll discuss in this unit that does not involve the use of some derived library. It's a simple enough way to use Redis that one isn't particularly needed (though when you compare it to ASP.NET Core you'll see why a framework integration can be really nice).

In this section, we'll walk through an application with the same model as the ASP.NET Core example, the major differences are that instead of using EntityFramework Core, we use EntityFramework, and instead of using ASP.NET Core WebAPI we are using the ASP.NET API scaffolding, consequentially the structure of the projects are quite different.

## **Additional Prerequisites for this section**

* Because this is an ASP.NET section it necessary for you to have a Windows Machine (preferably Windows 10+) to work on. The reality of the .NET Framework is that it only runs on Windows, so you'll need Windows.
* You must have the [.NET Framework 4.8.1 SDK installed](https://dotnet.microsoft.com/en-us/download/visual-studio-sdks)
* The example here uses MS SQL Server Express's [LocalDB](https://learn.microsoft.com/en-us/sql/database-engine/configure-windows/sql-server-express-localdb?view=sql-server-ver16" \t "_blank).
* It's highly recommended that you use either Visual Studio or Rider for this section.

# Hands-On Caching with ASP.NET

In this Hands-On, we'll be exploring how to use Caching in ASP.NET. You'll likely want to open up the csproj file and checkout the ***starting-point*** branch before proceeding, unless of course you'd prefer to build everything from scratch.

The csproj file is located at **/src/section\_3/section3.3/section3.3.csproj**. It's highly recommended that you use Visual Studio or Rider for this Hands-On.

## **Migrate the Database**

The first thing to be done is of course migrate the database to where we need it to be. In this step we'll create the database files, create the tables, and seed some data into the database. To do this, all you'll need to do is in the Package Manager Console run:

**update-database**

## **Create our Lazy Multiplexer**

Unlike ASP.NET Core, dependency injection is not exactly a given in ASP.NET, there are competing modes of handling dependency injection.

To simplify our example we'll be using a Lazy Multiplexer. In the file **Redis.cs**, add the following code to the Redis class:

**private static readonly Lazy LazyMuxer;**

**static Redis()**

**{**

**var options = new ConfigurationOptions**

**{**

**EndPoints = { "localhost:6379" },**

**Password = ""**

**};**

**LazyMuxer = new Lazy(() => ConnectionMultiplexer.Connect(options));**

**}**

**public static ConnectionMultiplexer Muxer => LazyMuxer.Value;**

**public static IDatabase Database => Muxer.GetDatabase();**

This will create the Multiplexer the first time it's referenced in our app, and provides a static way of accessing a single multiplexer throughout our application.

## **Caching Average Sale**

Unlike the ASP.NET Core example, we need to make use of the **ConnectionMultiplexer** and its methods directly and leverage the core Redis API. We can approximate that behavior pretty easily using the **StringGet** and **StringSet** methods that we explored earlier in this course. Find the ***// TODO Section 3.3 step 1*** mark in ***AverageController.cs*** and add the following bit.

**var db = Redis.Database;**

**var avg = (double?)await db.StringGetSetExpiryAsync($"average:{id}", TimeSpan.FromHours(1));**

**if(avg != null)**

**{**

**stopwatch.Stop();**

**return Ok(new Dictionary**

**{**

**{"average", avg.Value },**

**{"elapsed", stopwatch.ElapsedMilliseconds }**

**});**

**}**

This will check the cache, and if the average has already been computed, it will simply return the computed result. Using **StringGetSetExpiry** has the added effect of resetting the expiration for more frequently accessed averages.

Now, if we fail to find the average for the given employee, we'll need to compute it from our SQL database, just as we did in the previous example. After computing that, we'll use a simple **StringSet** to set the key and the expiration. Find the ***// TODO Section 3.3 step 2*** mark, and add the following to set the field in the cache, along with an expiration of an hour.

**await db.StringSetAsync($"average:{id}", avg, TimeSpan.FromHours(1));**

## **Caching our Top Salesperson**

For our Top Salesperson aggregation we have two keys that we're working with so our gets/sets will look slightly different. For the **StringGet** we will pass an array of keys, and unpack from the resultant **RedisValue[]** array.

Find the ***// TODO Section 3.3 step 3*** mark and add the following to do just that.

**var res = await db.StringGetAsync(new RedisKey[] { "top:sales", "top:name" });**

**long? topSales = (long?)res[0];**

**string topSalesName = res[1];**

**if(topSales.HasValue && !string.IsNullOrEmpty(topSalesName))**

**{**

**stopwatch.Stop();**

**return Ok(new Dictionary**

**{**

**{ "sum\_sales", topSales },**

**{ "employee\_name", topSalesName },**

**{ "time", stopwatch.ElapsedMilliseconds }**

**});**

**}**

Now, if we haven't run this calculation in the last 5 minutes, we'll have had a cache miss. In which case, we'll run the same calculation we did in the ASP.NET Core example, after which, we'll set the two relevant keys, along with a 5 minute expiration **TimeSpan**. We don't want to make multiple round trips to Redis, so we'll collect the dispatched tasks and await them.

Find the ***// TODO Section 3.3 step 4*** mark and add the following.

**var topSalesSetTask = db.StringSetAsync("top:sales", topSalesperson.sumSales, expiry: TimeSpan.FromMinutes(5));**

**var topNameSetTask = db.StringSetAsync("top:name", topSalesperson.Employee.Name, expiry: TimeSpan.FromMinutes(5));**

**await Task.WhenAll(topSalesSetTask, topNameSetTask);**

## **Cache Total Sales**

The last aggregation we'll look at here is the Total Sales Aggregation. We want this to expire this at the same time everyday (local midnight) of the next day.

Thus we'll start off in the ***Controllers/TotalSalesController.cs***'s **GetTotalSales** method's ***// TODO Section 3.3 step 5*** and add our normal check:

**long? totalSales = (long?) await db.StringGetAsync("totalSales");**

**if(totalSales != null)**

**{**

**return Ok(new Dictionary()**

**{**

**{ "Total Sales", totalSales.Value },**

**{ "elapsed", stopwatch.ElapsedMilliseconds }**

**});**

**}**

Then, after the ***// TODO Section 3.3 step 6*** mark, you can add the following - to have it cache the result, and set an expiration of the local midnight:

**var timeTillMidnight = DateTime.Today.AddDays(1) - DateTime.Now;**

**await db.StringSetAsync("totalSales", totalSales, timeTillMidnight);**

## **Run the App**

All that's left to do now is to run the app. To do that, hit F5 in Visual Studio or Rider, and it will launch the app.

Once it's running, you can visit the app at <http://localhost:50239/> and you can look at the result of our code at the following links:

* Total Sales: <http://localhost:50239/api/totalsales>
* Average: <http://localhost:50239/api/average/1>
* Total Sales: <http://localhost:50239/api/topsalesperson>

And you'll notice of course that after the initial load of each, that there's a significant increase in performance as the cache is hit rather than the database.

# 3.4 Distributed Session State in ASP.NET Core

In this Hands-On, we'll be exploring how to use Session State Management in ASP.NET Core. You'll likely want to open up the csproj file and check out the ***starting-point*** branch before proceeding, unless of course you'd prefer to build everything from scratch.

The csproj file is located at **/src/section\_3/section3.4/section3.4.csproj**.

Session state is the state maintained between the client's browser and a web server. However as we live in a world of increasingly larger distributed systems, we run into a problem. Do we really want the client to be dependent on a single, theoretically stateless application server?

The answer is no, yet this session-state can be absolutely critical for the functionality of some applications. Hence it's often really helpful to have a intermediate database act as a central holder for this session state. This pattern is so popular in fact that it's heavily integrated into both ASP.NET and ASP.NET Core.

Since Redis is so often tapped as the database acting as the distributed session state store, there are very straightforward integrations to add Redis as a native integration which we'll be exploring throughout the next couple of sections.

# Hands-On with Session State Management in ASP.NET Core

In this Hands-On, we'll be exploring how to use Session State Management in ASP.NET Core. You'll likely want to open up the csproj file and checkout the ***starting-point*** branch before proceeding, unless of course you'd prefer to build everything from scratch.

The csproj file is located at **/src/section\_3/section3.4/section3.4.csproj**.

It's highly recommended that you use Visual Studio or Rider for this Hands-On.

## **Review the State of the App**

The actual hands-on bit of this is really simple, so before we jump into it, let's take a look at the state of the current state of the application.

If you start the application with **dotnet run**, you'll see on the home page at <http://localhost:5097/> that the Session State is telling you the meaning of life is "42".

This is coming directly out of the Session State. If you look at the **Index** method of ***Controllers/HomeController.cs*** you'll see that we are setting the session state to 42 with:

**HttpContext.Session.SetString("meaning of life", fourtyTwo.ToString());**

Meanwhile, on the front-end, our view at ***Views/Home/Index.cshtml*** - is displaying the meaning of life key within the session with:

**@Context.Session.GetString("meaning of life")**

Finally, In ***Program.cs*** we have two lines of code:

**builder.Services.AddSession();**

The above line adds the Session Services to our available services.

**app.UseSession();**

The above adds the Session State Middleware to acquire the session state to be passed to the controllers that will interact with the session.

These are all the preliminaries for using **SessionState**. However, in this state, the session is not distributed, so the sessions will not be able to persist across different application servers, severely limiting your ability to scale out requests from an individual client.

## **Adding Distributed Session State Storage with Redis**

This part's really simple. Ahead of time, I've added:

**<ItemGroup>**

**<PackageReference Include="Microsoft.Extensions.Caching.StackExchangeRedis" Version="7.0.0" />**

**</ItemGroup>**

To the ***section3.4.csproj*** file, that's the exact same package we used earlier when we discussed using Redis as a Caching Layer for ASP.NET Core.

Enabling this as the backing for the session state is as simple as adding the Cache to our services container in that previous section. You only need to add the following to the services section of your ***Program.cs*** section at the mark ***// TODO Section 3.4 Step 1***

**builder.Services.AddStackExchangeRedisCache(options => options.ConfigurationOptions = new ConfigurationOptions{**

**EndPoints = { "localhost:6379" },**

**Password = ""**

**});**

And that's it, with that done, you can run the app again with **dotnet run**, and when you launch the application's site at <http://localhost:5097/>, the session will still work, the only difference is now the server is storing the session state in Redis instead of locally. You can even run **redis-cli monitor** to watch it work with the session state.

# 3.5 Distributed Session State with ASP.NET

ASP.NET has a well established Distributed Session State Provider for Redis called **Microsoft.Web.RedisSessionStateProvider**. It allows for a proper Integration of Session State with Redis, and similar to the ASP.NET Core case, is quite easy to get up and running with.

In this section, we'll be walking through a very simple application that leverages Session State, and see how to add Redis as a Provider of distributed session state in ASP.NET.

Because this is ASP.NET, this section is considered optional. None of the questions on the final exam will be derived from the material from this section. However, if you still work on non-core ASP.NET apps, this can still be quite useful, so if it's useful to you, I'd encourage you to walk through this tutorial.

## **Added Prerequisites for this Section**

* Because this is an ASP.NET section it necessary for you to have a Windows Machine (preferably Windows 10+) to work on. The reality of the .NET Framework is that it only runs on Windows, so you'll need Windows.
* You must have the [.NET Framework 4.8.1 SDK installed](https://dotnet.microsoft.com/en-us/download/visual-studio-sdks)
* It's highly recommended that you use either Visual Studio or Rider for this section.

# Hands-On with Session State in ASP.NET

In this Hands-On, we'll be exploring how to do Session State Management in ASP.NET. You'll likely want to open up the csproj file and checkout the ***starting-point*** branch before proceeding, unless of course you'd prefer to build everything from scratch.

The csproj file is located at **/src/section\_3/section3.5/section3.5.csproj**.

It's highly recommended that you use Visual Studio or Rider for this Hands-On.

## **State of the Application Pre-Redis**

Similarly to how Session State Management works in ASP.NET Core, ASP.NET has a very direct, native pattern for handling Session State, and a simple way to add Redis as the provider of the Distributed Session State.

First let's look at the state of the application without Redis as its Session State.

This application is simply a boiler plate ASP.NET MVC app, the template that ships out of the box when you create an MVC project with Visual Studio with a minor exception.

The contents of ***Views/Home/Index.cshtml*** have been replaced with:

**@{**

**ViewBag.Title = "Home Page";**

**}**

**<div class="jumbotron">**

**<h1>ASP.NET Distributed Session State with Redis</h1>**

**<p class="lead">Our Session Says the meaning of life is: @Session["the meaning of life"].</p>**

**</div>**

Which simply displays *the meaning of life* key in the Session State. Now, on the Controller side, updating the Session State is also quite trivial, you simply need to set the Session State key of your choosing. If you take a look at the **Index** method in ***Controllers/HomeController.cs*** you'll see a single line of code interacting with the Session State to set our *the meaning of life* key.

**Session["the meaning of life"] = 42;**

## **Adding Redis as the Session State Provider**

Adding Redis here is dead simple. First off you need to add the ***Microsoft.Web.Redis.RedisSessionStateProvider*** package to your application, we've already taken care of this for you.

Then in the ***Web.config*** file, you'll need to add the **sessionState** settings inside of the **<system.web>** tag, add the following below the ***<!-- Add Session State Here -->* mark in the *Web.config*** file.

**<sessionState mode="Custom" customProvider="MySessionStateStore">**

**<providers>**

**<add name="MySessionStateStore" type="Microsoft.Web.Redis.RedisSessionStateProvider" host="localhost" accessKey="" ssl="false" />**

**</providers>**

**</sessionState>**

After doing that, you can run the app by hitting F5, and when the browser launches (to <http://localhost:49982/>) you'll see that the session still works. The only difference being that Redis is now the provider for the Session State.

# Redis Stack and the Ad-Hoc API

Welcome to Unit 4 of RU102N. In this section we'll be taking a deeper look at Redis Stack.

Redis Stack enhances Redis with additional data structures and capabilities. These include:

* Probabilistic Data Structures
* The ability to ingest and query Time Series Data
* The ability to create and Query Graph data models with Cypher query language
* The ability to store and retrieve serialized JSON documents directly
* The ability to Query across hashes and JSON Documents

Unit 4 will focus on the first three, while Unit 5 will focus exclusively on the last two. We'll explore the client ecosystem surrounding Redis Stack, and how you can leverage the ad-hoc API provided by StackExchange.Redis to execute any command you need to against Redis.

# 4.1 The Ad-Hoc API

The Ad-Hoc API exposed by StackExchange.Redis is a powerful tool that allows you to execute Arbitrary commands against Redis.

Each of the Redis Stack libraries we'll be looking at in the forthcoming sections use this Ad-Hoc API to execute commands against Redis and to parse their results. In this section we'll look at how to run ad-hoc commands against Redis and parse their results.

We'll then use these ad-hoc commands to work with some of the Probabilistic Data Structures in Redis.

# Hands-On with Ad-Hoc Commands and Probabilistic Data Structures

In this Hands-On we'll be exploring how to use the Ad-Hoc API and Probabilistic Data Structures from StackExchange.Redis. You can either create your own project to follow along with and connect a multiplexer, or you can open **/src/section\_4/section4.1/section4.1.csproj** in your IDE.

The Ad-Hoc Commands are relatively straight forward to work with, they center entirely around two methods in StackExchange.Redis: **Execute** and **ExecuteAsync**. Naturally the primary differentiator between these two methods is whether or not they execute asynchronously. These methods each take three parameters:

* The command name
* A collection of argument objects for the command
* The Command Flags for the command

## **Command Name**

The Command Name is the name of the main command to be executed. If you look at the commands on the [Command Page of redis.io](https://redis.io/commands/) the Command Name is the first word preceding a space in the bold block, for any command with two or more words, only the first is a name, the remainder are technically options which must be passed in as arguments.

## **Arguments**

The arguments parameter can be passed one of two ways. Either using the params macro:

**await db.ExecuteAsync("SET", "foo", "bar");**

Or, you can pass them in as a collection:

**var arguments = new object[] { "foo", "bar" };**

**await db.ExecuteAsync("SET", arguments);**

Both work, the first way requires less code of course, but if you use it, you cannot pass any command flags in, since the params macro must be the last argument in the list.

## **Command Flags**

Command Flags provide some useful operational features, particularly around server selection and performance tuning. For example, if you are running a read-command, and do not want to hit your master shard, you can use the command flag **CommandFlags.DemandReplica**. There are corresponding flags to demand a master shard, prefer a master shard, or prefer a replica shard.

Another really useful thing you can drive via the command flags is to set a command to fire-and-forget. If you set a command to fire and forget, StackExchange.Redis will read back the result of your command, and release execution as soon as it sends the command to Redis, this can increase throughput, particularly in cases where you do not care about the result of a command.

## **Working with Probabilistic Data Structures**

Now that we know a little more about using the Ad-Hoc API, let's look a little bit deeper at how to use them with with something that we can't emulate with a better structured Library command from the main library.

Probabilistic Data Structures are a specialized family of data structures that allow you to approximate answers to very specific questions about a dataset. They're also great for an example as to how to use the ad-hoc API for something really useful. We're going to look at how to use two of these probabilistic data structures.

* **Bloom Filters**: Allows you to very compactly and quickly tell if an item has been added to a set yet.
* **Top-K**: This allows you to keep track of heavy hitters (most frequently seen items) in large streams of data.

## **Reserve and Add to a Bloom Filter**

A bloom filter allows you to quickly tell whether a word may have been added to a set in a space efficient way. Because false positives can occur, the bloom filter can tell if that something may be or is absolutely not in the set.

Let's demonstrate this by adding all the words from the book Moby Dick to our bloom filter in a relatively compact way. To add to a Bloom Filter, we just need arrange our words in such a way that they're in a sequential collection (e.g. an array or list) with the name of the Bloom Filter as the first element, and then each subsequent element being a word we want to add to the filter.

A simple aggregation will take care of this pretty after we've parsed out our words from a text document (this work is already done for you in the example).

**var bloomList = words.Aggregate(new List<object> { "bf" }, (list, word) =>**

**{**

**list.Add(word);**

**return list;**

**});**

With the words aggregated together into a much larger argument list, all that's left to do is use the **BF.RESERVE** command to reserve space for the Bloom Filter in Redis, specifying a key name, acceptable error rate and estimate of the size of our data set:

**await db.ExecuteAsync("BF.RESERVE", "bf", 0.01, 20000);**

Now we use the **BF.MADD** command to add everything to the Bloom Filter in one shot.

**await db.ExecuteAsync("BF.MADD", bloomList, CommandFlags.FireAndForget);**

## **Reserve and Add to Top-K**

The Top-K data structure lets us keep track of who the heavy hitters (most frequently seen items) are in enormous streams of data using the Heavy Keeper Data Structure. To Reserve a heavy keeper, all we need to do run the **TOPK.RESERVE** command:

**await db.ExecuteAsync("TOPK.RESERVE", "topk", 10, 20, 10, .925);**

And then we need to arrange all of our words so that they are not duplicated (this isn't absolutely necessary, but if you are using a cloud instance of Redis it will speed things up by saving on network bandwidth) and so that each word is followed by it's count. We'll run a couple quick aggregations against our word list to do this:

**// We need to organize the words into a list where each word is followed by the number of occurrences it has in Moby Dick**

**var topKList = words.Aggregate(new Dictionary<string, int>(), (dict, word) =>**

**{**

**if (!dict.ContainsKey(word))**

**{**

**dict.Add(word, 0);**

**}**

**dict[word]++;**

**return dict;**

**}).Aggregate(new List<object> {"topk"}, (list, kvp) =>**

**{**

**list.Add(kvp.Key);**

**list.Add(kvp.Value);**

**return list;**

**});**

Then we just need to call **TOPK.INCRBY** passing in our list of word/count pairs:

**// Add everything to the Top-K**

**await db.ExecuteAsync("TOPK.INCRBY", topKList, CommandFlags.FireAndForget);**

## **Querying our Bloom Filter**

Querying the Bloom filter is really simple, you just need to call **BF.EXISTS**. This will return a **RedisResult**, which is a highly versatile class that StackExchange.Redis uses to abstract the complexity of how the RESP protocol works. Fundamentally what it does is pulls in the result, and those results can be converted explicitly (by casting) to whatever the pertinent result type is. For **BF.EXISTS**, Redis will return an integer result. So let's try to work with this a bit. Let's check if the word "the" appears in Moby Dick (hint: it does).

**var doesTheExist = await db.ExecuteAsync("BF.EXISTS", "bf", "the");**

There's two things you can do with a **RedisResult**, you can cast/use it as its appropriate type: supported types are basically any value type, arrays, and as a dictionary of strings and RedisResults. If the type is supported by what was returned, and you try to cast it to that type, the cast will be successful, so let's try casting it to a couple of different types:

**var doesTheExistAsInt = (int)doesTheExist;**

**Console.WriteLine($"Typeof {nameof(doesTheExistAsInt)}: {doesTheExistAsInt.GetType()}");**

**var doesTheExistAsDouble = (double)doesTheExist;**

**Console.WriteLine($"Typeof {nameof(doesTheExistAsDouble)}: {doesTheExistAsDouble.GetType()}");**

Both of these are valid casts for our **RedisResult**, so it works just fine, but if we tried to cast it to something inappropriate (e.g. an array of **RedisResults**), it will fail with an **InvalidCastException**.

The determination as to whether or not a cast is valid is based off what Redis actually returned as a result of the command. You can check the **RedisResult** to see what the appropriate type is if you are unsure use the **Type** property, but the results will follow the format laid out in the [Command Pages on redis.io](https://redis.io/commands" \t "_blank) so you should have a pretty good idea ahead of time as to what you are looking for.

## **Enumerate the Top-K**

To Enumerate our Top-K, we just need to call **TOPK.LIST**. The trick here however, is that we need to cast it to something useable, like an array, preferably an array of strings.

To do that, we just need to perform an explicit cast, and then select each as a string.

**var res = await db.ExecuteAsync("TOPK.LIST", "topk");**

**var arr = ((RedisResult[])res!).Select(x=>x.ToString());**

**Console.WriteLine($"Top 10: {string.Join(", ", arr)}");**

Finally, if we wanted the counts as of our Top-K as well (which will come back as a dictionary of values), we can easily reduce it to a dictionary, and convert that dictionary to an enumerable that we can use to print out a discernable list of values with their counts using the **ToDictionary** method.

**var withCounts = (await db.ExecuteAsync("TOPK.LIST", "topk", "WITHCOUNT")).ToDictionary().Select(x=>$"{x.Key}: {x.Value}");**

**Console.WriteLine($"Top 10, with counts: {string.Join(", ", withCounts)}");**

# 4.2 Time Series With .NET

Redis Stack supports a Time Series Data Structure, which allows your application to ingest and utilize time tagged numeric data in a series. There are three different modes that you can use to interact with the Time Series data structure in Redis.

* NRedisTimeSeries: A specialized library based on StackExchange.Redis
* The Ad-Hoc API
* NRedisStack: A newer library based on StackExchange.Redis which provides access to all the commands in stack

In this section we'll be looking at how to use NRedisTimeSeries to interact with the Time Series Data Structure in Redis.

# Hands-On with Redis Time Series

In this Hands-On we'll be exploring how to use the Time Series data structure in Redis using NRedisTimeSeries. You can either create your own project to follow along with and connect a multiplexer, or you can open **/src/section\_4/section4.2/section4.2.csproj** in your IDE.

## **Connecting to Redis**

NRedisTimeSeries is an extension of StackExchange.Redis, consequentially connecting to Redis is exactly the same with NRedisTimeSeries as with StackExchange.Redis. The key difference, is that at the top of your ***Program.cs*** file, you need to add a few imports to use the NRedisTimeSeries bits:

**using NRedisTimeSeries;**

**using NRedisTimeSeries.Commands.Enums;**

**using NRedisTimeSeries.DataTypes;**

NRedisTimeSeries is built as static extensions to IDatabase, so after you've gotten a handle to your IDatabase, you'll be able to execute whatever you like against the Time Series Data Structure.

## **Create Time Series and Compaction Rules**

A Time Series is a time-tagged sequence of numbers that flow into the same key. For our example here, we'll be working with a very simple Time Series ***sensor***. We'll have a producer task to add random data to the Time Series continually, we'll then have compaction rules that will find the average, min, and max that occurred in the Time Series during a given interval. First we'll create the Time Series.

**await db.TimeSeriesCreateAsync("sensor", 60000, new List{new TimeSeriesLabel("id", "sensor-1")});**

With the Time Series created, we'll then identify the three aggregations we want to run. We'll create a Time Series for each of our compactions, those Time Series will carry a label with them so that we can filter on that label later when we run our query.

After the Time Series are created, we'll then bind a rule to our primary Time Series, to run the compaction into the relevant Time Series. To do that we just need to call **TimeSeriesCreate** and **TimeSeriesCreateRule** for each rule we want to create:

**var aggregations = new TsAggregation[]{TsAggregation.Avg, TsAggregation.Min, TsAggregation.Max};**

**foreach(var agg in aggregations)**

**{**

**await db.TimeSeriesCreateAsync($"sensor:{agg}", 60000, new List<TimeSeriesLabel>{new ("type", agg.ToString()), new("aggregation-for", "sensor-1")});**

**await(db.TimeSeriesCreateRuleAsync("sensor", new TimeSeriesRule($"sensor:{agg}", 5000, agg)));**

**}**

## **Produce Time Series Data**

Time Series Data can consisted of any time-tagged numeric data, this can be anything from sensor temperature readings, to live stock prices from a stock exchange, to radio signals. In our example, we're going to have a producer task which produces a simple random integer every second.

To do this, all you need to do is use the **TimeSeriesAdd** method, you can either pass in a **Timestamp** or you can just pass in the string "***\****" - which will create the millisecond timestamp based on when the data reached Redis.

**var producerTask = Task.Run(async()=>{**

**while(true)**

**{**

**await db.TimeSeriesAddAsync("sensor", "\*", Random.Shared.Next(50));**

**await Task.Delay(1000);**

**}**

**});**

## **Primary Consumer**

Because we have the primary Time Series, and the extra compaction Time Series running in parallel, we will have two consumer tasks. The first of these will use **TimeSeriesGet** every second, to retrieve the most recent data from the Time Series.

**var consumerTask = Task.Run(async()=>{**

**while(true)**

**{**

**await Task.Delay(1000);**

**var result = await db.TimeSeriesGetAsync("sensor");**

**Console.WriteLine($"{result.Time.Value}: {result.Val}");**

**}**

**});**

## **Aggregation Consumer**

The second Consumer will retrieve the aggregated data across our compacted Time Series. This will run every 5 seconds to coincide with our compaction rules. In this case we'll use **TimeSeriesMGet**. In this case however we'll be using the label that we created earlier to query multiple Time Series with the relevant label.

**var aggregationConsumerTask = Task.Run(async()=>**

**{**

**while(true)**

**{**

**await Task.Delay(5000);**

**var results = await db.TimeSeriesMGetAsync(new List<string>(){"aggregation-for=sensor-1"}, true);**

**foreach(var result in results)**

**{**

**Console.WriteLine($"{result.labels.First(x=>x.Key == "type").Value}: {result.value.Val}");**

**}**

**}**

**});**

## **Add a ReadKey and Run**

The last thing we need to do for this sample is to add a **Console.ReadKey()** at the bottom of our file. This will cause the tasks we spun up earlier to run endlessly until you press any key to exit. To run the example, just use **dotnet run** from your terminal.

# 4.3 Redis Graph

Another feature of Redis Stack is the ability to work with Graph Databases. There are a few ways that you can interact with Graph Databases in Redis Stack:

* NRedisGraph: another extension library based on StackExchange.Redis wrapping the Redis Graph Commands
* You can use the Ad-Hoc API to send Graph Commands Directly
* NRedisStack: A newer library based on StackExchange.Redis which provides access to all the commands in Stack

In this section, we'll work with NRedisGraph to run our commands and use the results of some basic Graph Queries.

# Hands-On with Redis Graph

In this Hands-On we'll be exploring how to use Graphs in Redis using NRedisGraph. You can either create your own project to follow along with and connect a multiplexer, or you can open **/src/section\_4/section4.3/section4.3.csproj** in your IDE.

## **Initialize Graph Object**

The command interface of NRedisGraph revolves around the **RedisGraph** class, which exposes the relevant RedisGraph commands. Consequentially the first thing we need to do, after we've initialized our multiplexer and gotten our **IDatabase** is to Initialize a new **RedisGraph** object, we'll also delete our ***pets*** graph (which is what we'll be working with in this example):

**var graph = new RedisGraph(db);**

**db.KeyDelete("pets");**

## **Create Our Graph**

In this example, we'll be working on a simple ***pets*** graph which will map animals to humans. We'll create two Humans (Alice, and Bob) and one dog (Honey).

RedisGraph is primarily built around the **GRAPH.QUERY** Command, which simply takes a key (our graph name) and a [Cypher Query](https://opencypher.org/" \t "_blank) and runs the cypher against the graph in the key, to produce a result. To create a person we'll use a cypher query with a create command:

**var createBobResult = await graph.QueryAsync("pets", "CREATE(:human{name:'Bob',age:32})");**

This creates a human (Bob), and returns a **ResultSet** that contains the results of the query. Since it's a create query, it doesn't respond with much, aside from the statistics from the query, how many nodes it created, how many properties it set, how many labels it added, and how long the query took to execute. You can print out all that information by accessing the **ResultSet**.

**Console.WriteLine($"Nodes Created:{createBobResult.Statistics.NodesCreated}");**

**Console.WriteLine($"Properties Set:{createBobResult.Statistics.PropertiesSet}");**

**Console.WriteLine($"Labels Created:{createBobResult.Statistics.LabelsAdded}");**

**Console.WriteLine($"Operation took:{createBobResult.Statistics.QueryInternalExecutionTime}");**

Let's add one more Human to our graph (Alice), with a similar query:

**await graph.QueryAsync("pets", "CREATE(:human{name:'Alice',age:30})");**

Finally we'll add a pet to our graph: Honey, a 5 year old Greyhound:

**await graph.QueryAsync("pets", "CREATE(:pet{name:'Honey',age:5,species:'canine',breed:'Greyhound'})");**

## **Add Some Relationships**

A graph database consists of two primal types, Nodes and Edges, just like a graph. Nodes tend to be nouns, so in our example here our pet, Honey, and humans, Alice and Bob are all nodes.

The Edges of our graph will tend to be verbs, so Honey might have an owner, so we can create an "OWNS" relationship between Honey and one of our Humans, let's make Honey's owner Bob, to do that we just match our pet and human and create a relationship between the nodes:

**await graph.QueryAsync("pets",**

**"MATCH(a:human),(p:pet) WHERE(a.name='Bob' and p.name='Honey') CREATE (a)-[:OWNS]->(p)");**

We can also create relationships between Honey and Both Alice and Bob to make them walkers of Honey:

**await graph.QueryAsync("pets",**

**"MATCH(a:human),(p:pet) WHERE(a.name='Alice' and p.name='Honey') CREATE (a)-[:WALKS]->(p)");**

**await graph.QueryAsync("pets",**

**"MATCH(a:human),(p:pet) WHERE(a.name='Bob' and p.name='Honey') CREATE (a)-[:WALKS]->(p)");**

## **Querying**

Now that we've created some nodes and relationships in our Graph we can go about querying them. To run a Match, we'll use the **Query** command, as we've been using in the previous examples, however in this case we'll be specifying a return type. So let's start off by finding all the owners of Honey. To perform the query, perform a match between a human and pet, where there is an own relationship between the human and the pet and the pet's name is honey, from this we'll return the human.

**var matches = await graph.QueryAsync("pets", "MATCH(a:human),(p:pet) where (a)-[:OWNS]->(p) and p.name='Honey' return a");**

You can then pull out the first record from that result and print out that record, and you'll get the value of Bob (which is what we were expecting).

**var record = matches.First();**

**Console.WriteLine($"Honey's owner nodes: {record}");**

We can then query all of the walkers of Honey by pulling back all of the human nodes that have an ***WALKS*** relationship with Honey. If we do a little bit more introspection on the result set, we can pull out individual nodes from each record, and print out only the information that we need, e.g. a name to print out the people who walk Honey.

**matches = await graph.QueryAsync("pets", "MATCH(a:human),(p:pet) where (a)-[:WALKS]->(p) and p.name='Honey' return a");**

**foreach (var rec in matches)**

**{**

**var node = (Node)rec.Values.First();**

**Console.WriteLine($"{node.PropertyMap["name"].Value} walks honey");**

**}**

In reverse, we can also enumerate all the dogs owned by a particular human by matching that same owns relationship with that human's name and with a pet who's species is "canine":

**matches = await graph.QueryAsync("pets", "MATCH(a:human),(p:pet) where (a)-[:OWNS]->(p) and p.species='canine' and a.name='Bob' return p");**

**foreach (var rec in matches)**

**{**

**var dogs = rec.Values.Select(x=>(Node)x).Select(x=>x.PropertyMap["name"].Value.ToString());**

**Console.WriteLine($"Bob's dogs are: {string.Join(", ", dogs)}");**

**}**

# 5 Redis OM

n this unit, we'll be covering the last two primary use cases of Redis Stack:

* Storage and Retrieval of serialized JSON documents
* Indexing and Querying JSON documents stored in Redis

We'll be learning how to do all of this using [Redis OM .NET](https://github.com/redis/redis-om-dotnet" \t "_blank), an Object Mapping library that facilities the storage and retrieval of your classes in Redis using LINQ.

# 5.1 Modeling Objects with Redis OM

Redis OM .NET gives you the ability to natively model your objects in Redis. Let's start off with the model that we will use for the remainder of this course. It will look very similar to the model that we used in Unit 3, but with some important changes:

**public class Employee**

**{**

**public string? Id { get; set; }**

**public List<string>? Sales { get; set; }**

**public Address? Address { get; set; }**

**public string? Name { get; set; }**

**public int Age { get; set; }**

**}**

**public class Sale**

**{**

**public string? Id { get; set; }**

**public int Total { get; set; }**

**public Address? Address { get; set; }**

**}**

**public class Address**

**{**

**public string? StreetAddress { get; set; }**

**public string? PostalCode { get; set; }**

**public GeoLoc Location { get; set; }**

**public Address? ForwardingAddress { get; set; }**

**}**

In the examples for the remainder of this course, this model will be found in ***Model.cs***, and will need to be imported into ***Program.cs*** before being used.

In this section, we'll learn how to take this model, and turn it into something that can be stored and queried in Redis.

## **The Document Attribute**

The **DocumentAttribute** is the primary attribute for a class, it's meant to decorate the class definition directly. It defines a few things with regards to how Redis OM will store and index a document inserted into Redis. The **DocumentAttribute** need only decorate a class that will be inserted into Redis, it does not need to decorate classes that will be embedded within the documents that we are inserting.

In our example, the only classes we'll be inserting are the **Employee** and **Sale** class, the **Address** will be embedded within it. With the **DocumentAttribute** we define the parameters of the class. This includes the following:

* Storage Type
* Key prefix
* Index Name

### Storage Type

Redis OM has two ways of storing documents: as JSON and as Hashes. It's highly recommended that you use JSON, unless there's some compelling reason not to do so. JSON is better at storing more complicated objects, particularly collections within those objects.

### Key Prefix

The Key Prefix is expressed as a collection of potential prefixes that keys inserted into Redis can hold. When the index is created later this will limit what redis will index to keys matching the given prefix. During key-generation at insertion time, if the Prefixes array is set, Redis OM will select the first prefix and use that as the prefix for all inserted keys. If no prefixes are set, Redis OM will use the fully qualified class name as the prefix for keys when it inserts your documents into Redis.

### Index Name

The Index Name is the name of the index in Redis. An index is the construct that will allow us to query records by their values later with LINQ. The Index Name has an easily computable name if not set (**<lowercaseclassname>-idx**). While it's not critical at all that this be set, you should set it if you want to give your index a specific name.

## **The Redis Id Field Attribute**

The **RedisIdFieldAttribute** marks the Id Field for the model. There should only be one of these per class, on the top level class being stored. This Id Field is attached to at storage time. If there is an ID present, then the Id that's in that field is respected and used during key generation, if it's empty, then Redis OM sets it based off the Id Generation Strategy, which can be changed in the **DocumentAttribute**, and by default will use a ULID. It's recommended that the Id Field be either a **string**, **Ulid**, or **Guid**, unless you provide a generation strategy, or supply the ID itself when creating the classes for insertion.

## **The Indexed Attribute**

The **IndexedAttribute** is the workhorse attribute of Redis OM, it can be applied to most types, strings, enums, numerics, **GeoLoc**, and even other classes. The **IndexedAttribute** tells Redis OM that you want to be able to query the field, or in the case of other classes - sub-fields of those classes.

The **IndexedAttribute** allows you to mark fields as sortable, and for more complex documents allows you to specify either how far down the object tree the indexing will cascade, or the exact paths you want to search on.

## **The Searchable Attribute**

The **SearchableAttribute** enables full-text search on a string field stored in Redis. If an attribute is marked as searchable, ***==*** operations in LINQ will be tantamount to asking for a match against the Full Text Search field rather than an exact match.

# Hands-On Modeling with Redis OM .NET

In this Hands-On we'll be exploring how to do object and index modeling with Redis OM.NET.

You can either create your own project to follow along with and connect a **RedisConnectionProvider**, or you can open **/src/section\_5/section5.1/section5.1.csproj** in your IDE.

## **Declare Indexes with DocumentAttribute**

After we've written out the model for our class, we need to start decorating it. The first attribute we will decorate our classes with is the **DocumentAttribute** where we can set the class level attributes for the index such as the prefix, index name, and storage type.

Find the class definition for **Employee** in **Model.cs** and add the following above it:

**[Document(StorageType = StorageType.Json, Prefixes = new []{"Employee"}, IndexName = "employees")]**

**public class Employee**

This will tell Redis OM that we want the Employee stored as a JSON document, that we want the prefix that the key-generator uses to be "**Employee**", and that the index name that Redis OM will use to index the employees will be simply "**employees".**

Now with the **Sale** class, we will make our **DocumentAttribute** declaration a bit simpler:

**[Document(StorageType = StorageType.Json)]**

**public class Sale**

This will make the Sale class be stored as JSON, however Redis OM will revert to its defaults for Index Naming and Prefix generation. In this case, the Index Name will simply be "**sale-idx**", and the prefix used will be the fully qualified class name of the model class. In this case that will be ***section5.\_1.Sale:*** but this will vary based on your namespace.

## **Add Id Fields**

Adding an Id field to your model is not strictly speaking necessary, but it's highly recommended that you have one. Without an ID field, you will have some restrictions on how you can update and delete your objects.

To Add an Id field to a class just mark the field with the **RedisIdFieldAttribute**. In our case, both our Employee and Sale classes have a field called **Id** so each should look like this after they are marked:

**[RedisIdField]**

**public string? Id { get; set; }**

## **Indexing Scalars**

Indexing Scalar Values with Redis OM is really simple, all you need to do is mark the value as **Indexed** or **Searchable** depending on what kind of matching you want to do on the value. Basically, as long as you aren't looking to do full-text search on a field, marking it as **Indexed** will be sufficient.

In our model, the only field we'll mark as full-text searchable will be the **StreetAddress** field, we'll mark all the other fields as indexed. Keep in mind, that you do not actually have to mark fields as Indexed to store them, it's just that if you do not mark them as indexed, you will not be able to run searches on those fields.

After we've run through and indexed all our scalar fields, the model will look like this:

**[Document(StorageType = StorageType.Json, Prefixes = new []{"Employee"}, IndexName = "employees")]**

**public class Employee**

**{**

**[RedisIdField]**

**[Indexed]**

**public string? Id { get; set; }**

**public List<string>? Sales { get; set; }**

**public Address? Address { get; set; }**

**[Indexed]**

**public string? Name { get; set; }**

**[Indexed]**

**public int Age { get; set; }**

**}**

**[Document(StorageType = StorageType.Json)]**

**public class Sale**

**{**

**[RedisIdField]**

**[Indexed]**

**public string? Id { get; set; }**

**[Indexed]**

**public string? EmployeeId { get; set; }**

**[Indexed]**

**public int Total { get; set; }**

**public Address? Address { get; set; }**

**}**

**public class Address**

**{**

**[Searchable]**

**public string? StreetAddress { get; set; }**

**[Indexed]**

**public string? PostalCode { get; set; }**

**[Indexed]**

**public GeoLoc Location { get; set; }**

**public Address? ForwardingAddress { get; set; }**

**}**

## **Indexing Embedded Documents**

An embedded document is a complex object, e.g. our **Employee** and **Sale** classes have a **Address** property. There are two methods for indexing embedded documents with Redis OM.

* Via JSON Path, explicitly listing the scalars within the object graph you want to index.
* Cascading into the object graph to a certain depth.

### Index Precisely by JSON Path

You can index fields precisely by their JSON path by using the **JsonPath** property of the **IndexedAttribute**. So for example, if we only wanted to index the **Location** and **PostalCode** of our employees, we can do so by marking their Address field like so:

**[Indexed(JsonPath = "$.Location")]**

**[Indexed(JsonPath = "$.PostalCode")]**

**public Address? Address { get; set; }**

### Indexing By Cascading

If you want to have your embedded documents more or less indexed as if they would be if they were the top-level Item, you can set a **CascadeDepth** on each field. This will traverse the object graph to the specified depth, and index everything accordingly.

For instance, if we wanted to index the **ForwardingAddress** in our **Address** field, we can by setting a cascade depth on that field to 1. This will prevent it from cascading deeper than just the scalar attributes beneath the address field:

**[Indexed(CascadeDepth = 1)]**

**public Address? ForwardingAddress { get; set; }**

Of course, we aren't actually indexing the **Address** directly, so we will need to set a cascade depth on the **Sale** class's **Address** field to 2 for it to fully propogate down.

**[Indexed(CascadeDepth = 2)]**

**public Address? Address { get; set; }**

## **Indexing Collections**

Arrays of scalar strings can be indexed with Redis OM (support for other types of scalars is coming shortly), to index them - e.g. the **Sales** field in **Employee** - you just need to mark the field as Indexed. This will work on both arrays and lists.

**[Indexed]**

**public List? Sales { get; set; }**

## **Connect to Redis**

Now that we've modeled our data and how we want it indexed, the last thing we need to do is create our indexes in Redis. To do that, we'll first need to initialize a **RedisConnectionProvider**.

The **RedisConnectionProvider** provides connected objects to Redis, specifically a **RedisConnection** (somewhat lower-level command interface), as well as the **RedisCollection<T>** and **AggregationSet<T>** which we will explore more in depth in later sections.

To connect to Redis, you simply need a [Redis URI](https://github.com/redis-developer/Redis-Developer-URI-Spec/blob/main/spec.md" \t "_blank), which looks something along the lines of ***redis://localhost:6379*** if you are connecting locally, or ***redis://:password@host:port*** if you are connecting to a cloud instance. You'll see the following in ***Program.cs***, which is a provider connected to a local instance of Redis:

**var provider = new RedisConnectionProvider("redis://localhost:6379");**

## **Create our Index**

Creating an index is very simple, you just need to call the **IRedisConnection** **CreateIndex** method, passing in the type that you want to create the index for.

**await provider.Connection.CreateIndexAsync(typeof(Sale));**

**await provider.Connection.CreateIndexAsync(typeof(Employee));**

## **A note on the Index Deletions**

In our example we also have a call to **DropIndexAndAssociatedRecords**, which is a method that deletes the index and deletes all the Records associated with it. This is purely for the sake of consistency between our demos and should only be done if necessary in your own code. In general, if you want to drop an index, you can do so with **DropIndex**, which will not delete the indexed records.

## **Run our example**

All you need to do to run this example now that you've updated the connection logic for your use case is call **dotnet run** in your terminal. This will create the indexes in Redis for you.

# 5.2 Inserting Documents

Redis OM enables you to insert your modeled objects into Redis. As we'll find out in this section, doing so is very straightforward, but there are some considerations you'll want to keep in mind as you are inserting your objects into Redis.

# Hands-On with Object Insertion with Redis OM .NET

In this Hands-On we'll be exploring how to insert objects into Redis with Redis OM .NET. You can either create your own project to follow along with and connect a RedisConnectionProvider, or you can open **/src/section\_5/section5.2/section5.2.csproj** in your IDE.

## **The state of our Model**

You can observe our model as we left it off in the previous section in ***Model.cs***. Practically every field is indexed, and the two types that we are going to insert today are the **Employee** and **Sale** types. They each have a **RedisIdField** decorated **Id** field, that will be used for handling the model's id, we'll take a look at what that means in a bit.

Both classes have a **StorageType** of JSON, meaning they'll be serialized as JSON as they are inserted into Redis. The primary difference in the way they are decorated is that the **Employee** class sets what its index name is, and tells Redis OM what prefixes it's allowed to use when indexing. The **Sale** class on the other hand allows Redis OM to use its defaults.

## **A Simple Insertion**

Inserting an object into Redis OM is simple, you use the **IRedisCollection<T>.Insert** method. You grab a handle to **IRedisCollection<T>** by using the **RedisCollection<T>**, then it's as simple as inserting your objects. Let's try to insert an employee.

**var employees = provider.RedisCollection();**

**var employee = new Employee**

**{**

**Name = "Steve",**

**Address = new Address**

**{**

**StreetAddress = "Main Street",**

**PostalCode = "34739",**

**Location = new GeoLoc(-81.006, 27.872)**

**},**

**Sales = new List()**

**};**

**var key = employees.Insert(employee);**

Notice how we did not initialize an Id for this employee before we inserted it. If an Id is not provided, one will automatically be generated by Redis OM, by default, this will be a *[ULID](https://github.com/ulid/spec" \t "_blank)* - which is essentially a sortable UUID. This will be bound directly to the Id Field in the model.

Additionally, when you use Insert, it will respond with a key name, which is the prefix for the model with the Id added to the end, you can observe these behaviors by simply printing out the key and the id that was generated by the insertion:

**Console.WriteLine($"Employee Id: {employee.Id}");**

**Console.WriteLine($"Key Name: {key}");**

A sample output for this would be:

**Employee Id: 01GKMARM1N59CWSVKN4DPV9A6T**

**Key Name: Employee:01GKMARM1N59CWSVKN4DPV9A6T**

Now let's do something slightly different, lets insert a **Sale** into Redis, but this time we will use the **IRedisConnection.Set** method, and instead of having Redis OM generate an ID for us, we'll generate one ourself.

**var sale = new Sale**

**{**

**Id = Guid.NewGuid().ToString(),**

**Address = new Address**

**{**

**StreetAddress = "Pinewood Ave",**

**PostalCode = "10001",**

**Location = new GeoLoc( -73.991, 40.753)**

**},**

**EmployeeId = employee.Id,**

**Total = 5000,**

**};**

**key = provider.Connection.Set(sale, TimeSpan.FromMinutes(5));**

**Console.WriteLine($"Sale Id: {sale.Id}");**

**Console.WriteLine($"Key Name: {key}");**

This operates very similarly to how the **Insert** method did, with the key difference being that since **Sale** leans on the defaults for Redis OM, it uses the fully qualified class name as the key prefix.

Since we provided the ID in the form of a freshly generated GUID, Redis OM will not try to generate a ULID for you. Finally, since we added a **TimeSpan** as our second argument, we've ensured that after the allotted time (5 minutes), the key will expire automatically and be removed from Redis.

An example output for this would look like:

**Sale Id: 161f0348-7557-4d9f-994b-dbb06b500b0e**

**Key Name: section5.\_2.Sale:161f0348-7557-4d9f-994b-dbb06b500b0e**

# 5.3 Querying Documents in Redis OM

In this section, we'll be looking at how to query documents that have been inserted in to Redis using Redis OM. There are three ways to query our documents in Redis OM.

* Using The RedisCollection's LINQ API
* Directly using ID or Key Name
* Using the Low-Level **IRedisConnection.Search** API

In this section we'll primarily be focusing on LINQ, but we'll very briefly cover the other two ways here.

## **Querying with LINQ**

Querying with LINQ revolves around using the **IRedisCollection<T>**, this works like most **IQueryable** types where it collects predicates to build out the query and then materializes when enumerated (e.g. a call to **ToList** or **ToArray**). Supported LINQ methods include:

* Where
* Select
* OrderBy
* OrderByDescending
* Skip
* Take
* Any
* First
* FirstOrDefault
* Single
* SingleOrDefault
* Count

These methods work as you'd expect them to if you've ever used an IQueryable interface before. However, there are a couple of key caveats to make around the Selects and OrderBy predicates. Selects will work best with fields that have been indexed. OrderBy predicates will only work on fields that have been marked sortable. So if we go into our **Employee** model, we can update the **IndexedAttribute** above the Name property to set it to Sortable:

**[Indexed(Sortable = true)]**

**public string? Name { get; set; }**

## **Querying by Id or Key**

Probably the simplest way to query a document in Redis is to use its Id or Key Name. You can use either just the Id, or the complete key name, whichever is preferable for you. To query this you would just use the **IRedisCollection<T>.FindById** methods, passing in either the Id or Key, the Collection is smart enough to construct a key regardless of which you end up using in your query.

### Querying with the Raw Search API

You can still perform strongly typed searches using the Raw Search API, to do this you would invoke **IRedisConnection.Search<T>** method, this takes a **RedisQuery** object which you can initialize in line with the arguments in **[FT.SEARCH](https://redis.io/commands/ft.search/" \t "_blank)**.

By using the generic, you are able to pull back your results as strongly typed objects.

# Hands-On Querying Redis with Redis OM

In this Hands-On we'll be exploring how to query your documents in Redis with Redis OM .NET. You can either create your own project to follow along with and connect a RedisConnectionProvider, or you can open **/src/section\_5/section5.3/section5.3.csproj** in your IDE.

## **Make Age and Name Sortable**

In this section we'll only be working with Employees, the first thing we need to do is to make a minor update to the model making the name and age sortable, to do this, open ***Model.cs*** and change the **IndexedAttribute** in the Age and Name property so they look like this:

**[Indexed(Sortable = true)]**

## **Insert some records**

Let's now insert some records into Redis. We'll just hard code the 5 people we were using in Unit 3, but they will be slightly more complex now because they have some extra fields, e.g. Age and Address (with a geoloc!)

**var alice = new Employee**

**{**

**Name = "Alice",**

**Age = 45,**

**Address = new Address { StreetAddress = "Elm Street", Location = new GeoLoc(-81.957, 27.058), PostalCode = "34269" }**

**};**

**var bob = new Employee**

**{**

**Name = "Bob",**

**Age = 60,**

**Address = new Address() { StreetAddress = "Bleecker Street", Location = new GeoLoc(-74.003, 40.732), PostalCode = "10014" }**

**};**

**var charlie = new Employee**

**{**

**Name = "Charlie",**

**Age = 26,**

**Address = new Address() { StreetAddress = "Ocean Boulevard", Location = new GeoLoc(-121.869, 36.604), PostalCode = "93940" }**

**};**

**var dan = new Employee**

**{**

**Name = "Dan",**

**Age = 42,**

**Address = new Address() { StreetAddress = "Baker Street", Location = new GeoLoc(-0.158, 51.523), PostalCode = "NW1 6XE" }**

**};**

**var yves = new Employee**

**{**

**Name = "Yves",**

**Age = 19,**

**Address = new Address() { StreetAddress = "Rue de Rivoli", Location = new GeoLoc(2.361, 48.863), PostalCode = "75003" }**

**};**

**await employees.InsertAsync(bob);**

**await employees.InsertAsync(alice);**

**await employees.InsertAsync(charlie);**

**await employees.InsertAsync(dan);**

**await employees.InsertAsync(yves);**

## **Our First Query with a String**

For our first query, we'll be using the **First** method to find the first employee in Redis named "Bob", This operation works exactly how you would expect it would if you've used LINQ in the past:

**Console.WriteLine($"Employees Named Bob");**

**var alsoBob = await employees.FirstAsync(x=>x.Name == "Bob");**

**Console.WriteLine($"Bob's age is: {alsoBob.Age} and his postal code is: {alsoBob.Address!.PostalCode}");**

The **FirstAsync** method materializes immediately serializing the query in the pipeline and sending it onto Redis.

## **Numeric Range Queries**

To query a range, you simply need to use the greater than or less than operators against numeric values, as you would in any other LINQ context. Since we are querying multiple records here, we will use a **Where** predicate. Let's try querying all the people under forty:

**// Query by age**

**var employeesUnderForty = employees.Where(x => x.Age < 40);**

**Console.WriteLine("----Employees under 40----");**

**await foreach (var emp in employeesUnderForty)**

**{**

**Console.WriteLine($"{emp.Name} is {emp.Age}");**

**}**

## **Proximity Geo Queries**

One of the really neat features of Redis OM .NET is that it give you the ability to perform GeoFilters on your data in Redis. A Geo Filter is a radius filter from a point on Earth.

You can express a Geo Filter in terms of one of your model's attributes, for example the **Address.Location** attribute, which is a **GeoLoc**, a special type within Redis for holding a geographic position.

To run a Geo filter, simply provide the attribute as well as the desired center-point of your geo filter, the radius from the center point you want to search, and the units you want to use for the filter. For example, if we wanted to find all of the employees within 1500 miles of Philadelphia, we can do so like this:

**var employeesNearPhilly = await employees.GeoFilter(x=>x.Address!.Location, -75.159, 39.963, 1500, GeoLocDistanceUnit.Miles).ToListAsync();**

**Console.WriteLine("----Employees near Philly----");**

**foreach (var emp in employeesNearPhilly)**

**{**

**Console.WriteLine($"{emp.Name} lives in the postal code: {emp.Address!.PostalCode}");**

**}**

## **Sorting**

Any property that has been marked **Sortable = true** in the model definition can be sorted. To sort on a field, you will use an **OrderBy** and **OrderByDescending** predicate in constructing your query. So, if we wanted to select the names of of our employees in Ascending order, we could do so with an **OrderBy** and **Select**:

**var employeesByAge = await employees.OrderBy(x=>x.Age).Select(x=>x.Name!).ToListAsync();**

**Console.WriteLine($"In Ascending order: {string.Join(", ", employeesByAge)}");**

Conversely, if we wanted to order employees names in reverse alphabetical order we could do so with an **OrderByDescending**:

**var employeesInReverseAlphabeticalOrder = await employees.OrderByDescending(x=>x.Name).Select(x=>x.Name!).ToListAsync();**

**Console.WriteLine($"In Reverse Alphabetical Order: {string.Join(", ", employeesInReverseAlphabeticalOrder)}");**

# 5.4 Updating and Deleting Documents with Redis OM

In this section we'll look at how Redis OM performs the "U" and "D" of CRUD: Updates, and Deletes.

Unlike a traditional relational database Redis Stack does not have the the ability to natively query, update, and delete records in isolation on the Redis Server, so we need to use the client to do so. Fortunately, Redis OM makes these tasks pretty straightforward as we'll be exploring in this section.

# Hands-On Updating and Deleting Documents with Redis OM

In this Hands-On we'll be exploring how to Update and Delete Documents in Redis OM .NET. You can either create your own project to follow along with and connect a **RedisConnectionProvider**, or you can open **/src/section\_5/section5.4/section5.4.csproj** in your IDE.

## **Setting up our Example**

In this example, we'll be inserting two employees, Alice and Bob, and we'll add Sales to each of them, and then delete them both.

So let's start off by adding Alice and Bob to Redis, as we would any other document:

**var alice = new Employee**

**{**

**Name = "Alice",**

**Age = 45,**

**Address = new Address { StreetAddress = "Elm Street", Location = new GeoLoc(-81.957, 27.058), PostalCode = "34269" },**

**Sales = new List<string>()**

**};**

**var bob = new Employee**

**{**

**Name = "Bob",**

**Age = 60,**

**Address = new Address() { StreetAddress = "Bleecker Street", Location = new GeoLoc(-74.003, 40.732), PostalCode = "10014" },**

**Sales = new List<string>()**

**};**

**var bobKeyName = await employees.InsertAsync(bob);**

**await employees.InsertAsync(alice);**

## **Updating Documents with Redis OM**

Updating documents with Redis OM revolves around the **IRedisCollection**, which exposes two methods for this behavior: update and save. To perform any updates it's critical that the documents you are trying to update have an Id Field that has a value set, this allows Redis OM to determine what the key that it is updating should be.

### Update a single Document

Using **IRedisCollection.Update** allows you to run an update on a single record in Redis. So, if we wanted to add a bunch of sales for Bob, we would simply update the Bob record, and call **Update** on the Redis Collection:

**var sales = provider.RedisCollection<Sale>();**

**var saleInsertTasks = new List<Task<string>>();**

**var random = new Random();**

**for (var i = 0; i < 500; i++)**

**{**

**saleInsertTasks.Add(sales.InsertAsync(new Sale**

**{**

**Total = random.Next(1000, 30000),**

**EmployeeId = bob.Id**

**}));**

**}**

**await Task.WhenAll(saleInsertTasks);**

**bob.Sales.AddRange(saleInsertTasks.Select(x=>x.Result.Split(":")[1]));**

**await employees.UpdateAsync(bob);**

**var bobFromDb = employees.FindById(bob.Id!);**

**Console.WriteLine($"Bob has: {bobFromDb!.Sales!.Count} sales");**

## **Updating Multiple Documents**

You can update multiple documents simultaneously by using the **IRedisCollection.Save** method. When an **IRedisCollection** is enumerated, it maintains a snapshot of each record that it enumerates. If you update those records after they've been enumerated, and call **IRedisCollection.Save** on the collection, the collection will diff each record's current state against it's snapshot, and update the relevant documents accordingly.

So let's try this in practice, let's increment the age of each person whose name is "Alice":

**await foreach (var employee in employees.Where(x => x.Name == "Alice"))**

**{**

**Console.WriteLine($"Alice's old age: {employee.Age}");**

**employee.Age++;**

**}**

**employees.Save();**

**Console.WriteLine($"Alice's new age: {employees.First(x=>x.Name == "Alice").Age}");**

## **Deleting Documents**

There are two ways to delete Documents in Redis. If your model has an Id field, you can use the **IRedisCollection.Delete** method, passing in the document you want to delete. Redis OM will then remove the document from Redis for you.

So, let's just try removing Alice from Redis:

**await employees.DeleteAsync(alice);**

**Console.WriteLine($"Alice's present in Redis: {await employees.AnyAsync(x=>x.Name == "Alice")}");**

If our model does not have an Id field on it, we can also use the **IRedisConnection.Unlink** method, passing in the key name that you want to remove. When we inserted Bob before, we collected the key name from the insert, so let's use that to delete him from Redis:

**await provider.Connection.UnlinkAsync(bobKeyName);**

**Console.WriteLine($"Bob's present in Redis: {await employees.AnyAsync(x=>x.Name == "Bob")}");**

# 5.5 Aggregations with Redis OM

In this section we'll look at another really useful feature of Redis Stack and Redis OM, the ability to aggregate data within your documents together. Aggregations allow you to build pipelines for your documents to perform tasks such as:

* Query your Documents the same way you would with a **IRedisCollection**.
* Apply a variety of mathematical and string functions to your data.
* Group like documents together.
* Run Reductions against groups of documents.
* Further filter documents based off the results in your pipeline.

# Hands-On with Aggregations in Redis OM

In this hands-on we'll learn how to use aggregations with Redis OM .NET. This section will revolve around the **AggregationSet<T>** which is a construct of Redis OM meant to make Aggregations operate fluently.

We'll learn how to query things in Redis using aggregations, how to run functions against our data in Redis, how to group our records together by attributes, and how to run reductions on those groups of documents.

## **Minor Model Updates**

In order to use a document's field within an aggregation it must be either:

* Marked as sortable in the original index.
* Explicitly Loaded into the pipeline.

We'll go over both of these methodologies in this section, but so we don't become overly distracted by this, let's go into our model and set a couple of fields as sortable that we had previously left not sortable.

Specifically in the Sale class, we'll mark both the Total and EmployeeId fields as **Aggregatable** - this is an alias of **Sortable** - I like to use it because it makes it explicit that we're setting it this way so we can run aggregations on our field:

**[Indexed(Aggregatable = true)]**

**public string? EmployeeId { get; set; }**

**[Indexed(Aggregatable = true)]**

**public int Total { get; set; }**

## **The**AggregationSet<T>**and**AggregationResult<T>

The Aggregations pipeline is all built around the **AggregationSet<T>**, which is of course a generic that you can apply whatever your model is to. However, you'll notice that the return type of all the queries from the **AggregationSet<T>** is **AggregationResult<T>** rather than whatever the type **<T>** was that you passed into it.

This is critical to understanding how to construct queries within the **AggregationSet**. Aggregations are not meant to pull out model data from your database, rather they are meant to allow you to combine your document data within Redis, and reduce it down to some outputs of your pipeline. The **AggregationResult<T>** contains a **RecordShell**, which is always null when not in the pipeline, that **RecordShell** can be used to build expressions for our pipeline.

## **Simple Aggregations: Find Bob's Sales**

We've added some code to seed a bunch of sales data into our database. Using aggregations, we can find the sum of a given person's sales. We can do that by filtering the sales by those whose employee id is Bob's, and then running a summation on all those records.

**var saleAggregations = provider.AggregationSet(); // init aggregation set.**

**var sumBobSales = saleAggregations.Filter(x=>x.RecordShell!.EmployeeId == bob.Id).Sum(x=>x.RecordShell!.Total);**

**Console.WriteLine($"Bob's total sales: {sumBobSales}");**

# Apply Functions

One of the key features of Redis Aggregations is the ability to run apply functions against the data at rest in Redis. There are a variety of Apply Functions spanning five categories: Math, Strings, Time, Geographic, and Existence.

To run an apply function in Redis, you use the **AggregationSet.Apply** method, passing in a simple function you want to run against your data against Redis, as well as an alias you want to give to the output for each record you're running your apply against.

## **Math**

Apply functions support typical arithmetic operations you might run normally, as well as a subset of the **System.Math** static methods native to .NET. For example, if we were keeping tabs of our total sales for each employee, and we wanted to compute some Adjusted Sales figure for each employee based on a regional sales adjustment, and then round it up to the nearest integer we could do so like this:

**var employeeAggregations = provider.AggregationSet();**

**var adjustedSales = employeeAggregations.Apply(x=>Math.Ceiling(x.RecordShell.SalesAdjustment \* x.RecordShell.TotalSales), "ADJUSTED\_SALES");**

**foreach(var employee in adjustedSales)**

**{**

**Console.WriteLine($"Adjusted Sales: {employee["ADJUSTED\_SALES"]}");**

**}**

All of the normal mathematical operators are supported for .NET, the only major divergence is that the bitwise XOR operator "**^**", is reserved for power functions in the Aggregation Apply pipeline, and will not perform XORs See the table below for a list of supported mathematical apply functions:

### Supported Math Apply Functions

|  |  |  |  |
| --- | --- | --- | --- |
| **Function** | **Type** | **Description** | **Example** |
| Log10 | Math | Yields the 10 base log for the number | **Math.Log10(x["AdjustedSales"])** |
| Abs | Math | Yields the absolute value of the provided number | **Math.Abs(x["AdjustedSales"])** |
| Ceil | Math | Yields the smallest integer not less than the provided number | **Math.Ceil(x["AdjustedSales"])** |
| Floor | Math | Yields the smallest integer not greater than the provided number | **Math.Floor(x["AdjustedSales"])** |
| Log | Math | Yields the Log base 2 for the provided number | **Math.Log(x["AdjustedSales"])** |
| Exp | Math | Yields the natural exponent for the provided number (e^y) | **Math.Exp(x["AdjustedSales"])** |
| Sqrt | Math | Yields the Square root for the provided number | **Math.Sqrt(x["AdjustedSales"])** |

## **String Apply Functions**

There are also a variety of string based apply functions that are quite helpful if, for example, you wanted to craft a birthday message for each employee you could do so using string interpolation within the Apply pipeline:

**var birthdayMessage = employeeAggregations.Apply(x=>$"Happy Birthday {x.RecordShell.Name} you're now {x.RecordShell.Age} ","BIRTHDAY\_MESSAGE");**

**foreach (var employee in birthdayMessage)**

**{**

**Console.WriteLine($"{employee["BIRTHDAY\_MESSAGE"]}");**

**}**

### Supported String Functions

|  |  |  |  |
| --- | --- | --- | --- |
| **Function** | **Type** | **Description** | **Example** |
| ToUpper | String | Yields the provided string to upper case | **x.RecordShell.Name.ToUpper()** |
| ToLower | String | Yields the provided string to lower case | **x.RecordShell.Name.ToLower()** |
| StartsWith | String | Boolean expression - yields 1 if the string starts with the argument | **x.RecordShell.Name.StartsWith("bob")** |
| Contains | String | Boolean expression - yields 1 if the string contains the argument | **x.RecordShell.Name.Contains("bob")** |
| Substring | String | Yields the substring starting at the given 0 based index, the length of the second argument, if the second argument is not provided, it will simply return the balance of the string | **x.RecordShell.Name.Substring(4, 10)** |
| Format | string | Formats the string based off the provided pattern | **string.Format("Hello {0} You are {1} years old", x.RecordShell.Name, x.RecordShell.Age)** |
| Split | string | Splits the string with the provided string - unfortunately if you are only passing in a single splitter, because of how expressions work, you'll need to provide string split options so that no optional parameters exist when building the expression, just pass **StringSplitOptions.None** | **x.RecordShell.Name.Split(",", StringSplitOptions.None)** |

## **Geographic Apply Functions**

If you're using **GeoLoc** data structures within your model, you can also compute the distance from a different geo location using the **GeoDistance** apply function.

Let's take the Geo Loc of the Empire State Building (-74.0031713, 40.7484396). You can check the distance of all our employees from that landmark using the **ApplyFunctions.GeoDistance** method:

**var empireGeoLoc = new GeoLoc(-74.0031713,40.7484396);**

**var distanceFromEmpireStateBuilding = employeeAggregations**

**.Load(x=>new {x.RecordShell.Name})**

**.Apply(x=>ApplyFunctions.GeoDistance(x.RecordShell.Address.Location, empireGeoLoc),"DISTANCE\_FROM\_EMPIRE\_STATE\_BUILDING");**

**foreach (var res in distanceFromEmpireStateBuilding)**

**{**

**var employee = res.Hydrate();**

**Console.WriteLine($"{employee.Name} is {res["DISTANCE\_FROM\_EMPIRE\_STATE\_BUILDING"]} meters from the Empire State Building");**

**}**

# Grouping and Reduction

Redis OM also allows you to group records together by their attributes, and run reductions on the data within each group. Take our Top Salesperson example from Section 3. If you wanted to determine who the top salesperson was with our given model, you can group all of our sales by the employee ID, then run a summation on the sales, then order by the result of that summation:

**// Top Salesperson**

**var topSalesId = saleAggregations**

**.GroupBy(x=>x.RecordShell.EmployeeId)**

**.Sum(x=>x.RecordShell.Total)**

**.OrderByDescending(x=>x["Total\_SUM"])**

**.Take(1)**

**.First()["EmployeeId"]**

**.ToString();**

**var topSalesPerson = await employees.FindByIdAsync(topSalesId);**

**Console.WriteLine($"Top seller: {topSalesPerson.Name}");**

When running reductions, the name of the result in the aggregation pipeline is always the attribute's name that the reduction was being run on, plus the reduction command's name in screaming caps joined by an underscore. So in our example above the **SUM** of **x.RecordShell.Total** comes out to **Total\_SUM**. See the table below for a full enumeration of supported reducers and their command postfixes.

### Supported Reducers and Postfixes

|  |  |  |
| --- | --- | --- |
| **Command Name** | **Command Postfix** | **Description** |
| Count | COUNT | Number of records meeting the query, or in the group |
| CountDistinct | COUNT\_DISTINCT | Counts the distinct occurrences of a given property in a group |
| CountDistinctish | COUNT\_DISTINCTISH | Provides an approximate count of distinct occurrences of a given property in each group - less expensive computationally but does have a small 3% error rate |
| Sum | SUM | The sum of all occurrences of the provided field in each group |
| Min | MIN | Minimum occurrence for the provided field in each group |
| Max | MAX | Maximum occurrence for the provided field in each group |
| Average | Avg | Arithmetic mean of all the occurrences for the provided field in a group |
| StandardDeviation | STDDEV | Standard deviation from the arithmetic mean of all the occurrences for the provided field in each group |
| Quantile | QUANTLE | The value of a record at the provided quantile for a field in each group, e.g., the Median of the field would be sitting at quantile .5 |
| Distinct | TOLIST | Enumerates all the distinct values of a given field in each group |
| FirstValue | FIRST\_VALUE | Retrieves the first occurrence of a given field in each group |
| RandomSample | RANDOM\*SAMPLE\*{NumRecords} | Random sample of the given field in each group |