Demo: Connect an app to Azure Cache for Redis by using .NET Core

In this demo you will learn how to:

- Create a new Redis Cache instance by using Azure CLI commands.
- Create a .NET Core console app to add and retrieve values from the cache by using the **StackExchange.Redis** NuGet package.

Prerequisites

This demo is performed in Visual Studio Code (VS Code).

Create demo folder and launch VS Code

1. Open a PowerShell terminal in your local OS and create a new directory for the project.

```
md az204-redisdemo
```

2. Change in to the new directory and launch VS Code.

```
cd az204-redisdemo
code .
```

3. Open a terminal in VS Code and login to Azure.

```
az login
```

Create a new Redis Cache instance

1. Create a resource group, replace `<myRegion>` with a location that makes sense for you. Copy the first line by itself and edit the value.

```
$myLocation="<myRegion>"
az group create -n az204-redisdemo-rg -l $myLocation
```

2. Create a Redis Cache instance by using the `az redis create` command. The instance name needs to be unique and the script below will attempt to generate one for you. This command will take a few minutes to complete.

```
$redisname = "az204redis" + $(get-random -minimum 10000 -maximum 100000)
az redis create -l $myLocation -g az204-redisdemo-rg -n $redisname --sku Basic --
```

3. Open the Azure Portal (**https://portal.azure.com**) and copy the connection string to the new Redis Cache instance.
   - Navigate to the new Redis Cache.
   - Select **Access keys** in the **Settings** section of the Navigation Pane.
   - Copy the **Primary connection string (StackExchange.Redis)** value and save to Notepad.

Create the console application

1. Create a console app by running the command below in the VS Code terminal.

```
dotnet new console
```

2. Add the **StackExchange.Redis** NuGet package to the project.

```
dotnet add package StackExchange.Redis
```

3. In the *Program.cs* file add the `using` statement below at the top.

```
using StackExchange.Redis;
using System.Threading.Tasks;
```

4. Let's have the `Main` method run asynchronously by changing it to the following:

```
static async Task Main(string[] args)
```

5. Connect to the cache by replacing the existing code in the `Main` method with the following code. Set the `connectionString` variable to the value you copied from the portal.

```
string connectionString = "YOUR_CONNECTION_STRING";

using (var cache = ConnectionMultiplexer.Connect(connectionString))
{

}
```

✔ **Note:** The connection to Azure Cache for Redis is managed by the `ConnectionMultiplexer` class. This class should be shared and reused throughout your client application. We do *not* want to create a new connection for each operation. Instead, we want to store it off as a field in our class and reuse it for each operation. Here we are only going to use it in the **Main** method, but in a production application, it should be stored in a class field, or a singleton.

Add a value to the cache

Now that we have the connection, let's add a value to the cache.

1. Inside the `using` block after the connection has been created, use the `GetDatabase` method to retrieve an `IDatabase` instance.

```
IDatabase db = cache.GetDatabase();
```

2. Call `StringSetAsync` on the `IDatabase` object to set the key “test:key” to the value "some value". The return value from `StringSetAsync` is a `bool` indicating whether the key was added. Append the code below to what you entered in Step 1 of this section.

```
bool setValue = await db.StringSetAsync("test:key", "100");
Console.WriteLine($"SET: {setValue}");
```

Get a value from the cache

1. Next, retrieve the value using `StringGetAsync`. This takes the key to retrieve and returns the value. Append the code below to what you entered in Step 2 above.

```
string getValue = await db.StringGetAsync("test:key");
Console.WriteLine($"GET: {getValue}");
```

2. Build and run the console app.

```
dotnet build
dotnet run
```

The output should be similar to the following:

```
SET: True
GET: 100
```

Other operations

Let's add a few additional methods to the code.

1. Execute “PING” to test the server connection. It should respond with "PONG". Append the following code to the `using` block.

```
var result = await db.ExecuteAsync("ping");
Console.WriteLine($"PING = {result.Type} : {result}");
```

2. Execute “FLUSHDB” to clear the database values. It should respond with "OK". Append the following code to the `using` block.

```
result = await db.ExecuteAsync("flushdb");
Console.WriteLine($"FLUSHDB = {result.Type} : {result}");
```

3. Build and run the console app.

```
dotnet build
dotnet run
```

The output should be similar to the following:

```
SET: True
GET: 100
PING = SimpleString : PONG
FLUSHDB = SimpleString : OK
```

Clean up resources

When you're finished with the demo you can clean up the resources by deleting the resource group created earlier. The following command can be run in the VS Code terminal.

```
az group delete -n az204-redisdemo-rg --no-wait --yes
```