Azure Developer Immersion

Azure Redis Cache is a high-performance, memory-based cache which can significantly improve performance and scalability when you have data that needs to be accessed very frequently. It is accessed over the network, so it can be shared by multiple machines in a web farm if you choose to scale out your service. This makes it more flexible than simply caching data in a web server's memory; if you update a Redis cache entry, everything will have access to the new value. By using a Redis cache, you can take significant load off persistent storage systems such as SQL Azure.

Your web app currently reads the entire list of users every time it loads a web page. It does this so that it can populate an auto-completion list when you type "@" to refer to someone. The volume of information is manageable because groups are relatively small, but it is a good candidate for caching because it is usually the same and it is relatively easy to know when it should change. So you will cache it.

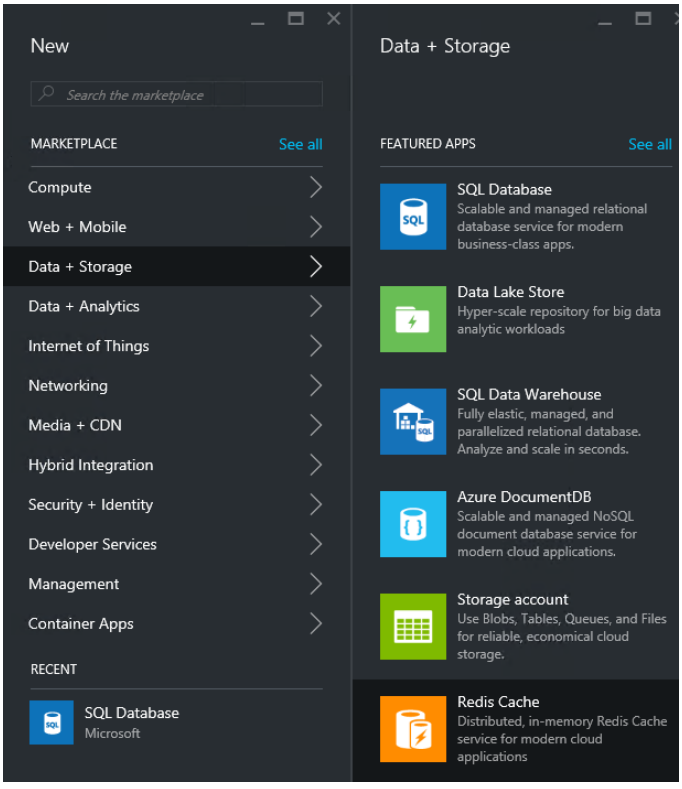
There is one exercise in this walkthrough:

1. Azure Redis Cache

Exercise 1: Azure Redis Cache

In this exercise, you will add Azure Redis Cache support to your application.

1. At this point you should be comfortable moving the next Task on your VSTS Task Board into the In Progress state. Do so now.
2. In the Azure portal, click **+ New** at the top left, then click **Data + Storage**. Select **Redis Cache**.



3. Enter a **DNS name** for the server. Azure requires this name to be globally unique.
4. Set the **Resource group** to **rGroup**.
5. Use the same **Location** that you have used so far.
6. Set the **Pricing tier** to **Basic**.
7. Click **Create**.

   While you are waiting for Azure to create the cache, let's modify the application to be able to use it.
8. Back in Visual Studio 2015, in the **Rg.ServiceCore** project's **Operations** folder, open **UserOperations.cs**.
9. At the bottom of the class, add this method, which we will fill in soon:

```
public static async Task RecacheAllUserListAsync(ApplicationDbContext dbContext)
{

}
```

   The plan is to call this every time something changes about a user, and rebuild the cached data.
10. In the **SetAvatarImage** method in the current class, just after the code calls the **dbContext.SaveChangesAsync()**, add a call to this new method:

```
await dbContext.SaveChangesAsync();
await RecacheAllUserListAsync(dbContext);

return HttpStatusCode.OK;
```

11. Make the same change to the **SetBannerImage** method.
12. In the **Rg.Api** project's **Controllers** folder, open **ProfileController.cs**. Find its **PostValidate** method. This sometimes creates new users. Find the code that does this (shown below) and change it by adding this highlighted code:

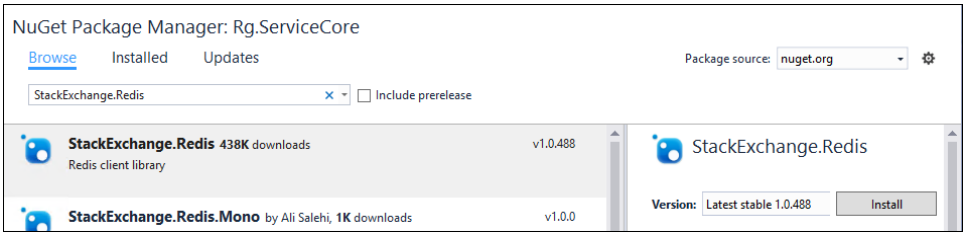
```
DbContext.UserInfos.Add(user);
}
await DbContext.SaveChangesAsync();
await UserOperations.RecacheAllUserListAsync(DbContext);
```

13. In the **Rg.Web** project's **Identity** folder, open **CustomUserStore.cs** and find the **CreateAsync** method. This also creates new users. Modify it by adding this highlighted code:

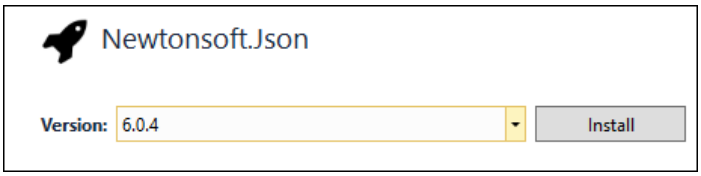
```
public async Task CreateAsync(ApplicationUser user)
{
    UserInfo entity = user.ToDbEntity();
    _db.UserInfos.Add(entity);
    await _db.SaveChangesAsync();
    await ServiceCore.Operations.UserOperations.RecacheAllUserListAsync(_db);
}
```

Now that we have ensured it is getting called every time a user is changed or added, you can implement the cache update method back in the **UserOperations.cs** file. To do this, you will need the Redis client library.

14. Right-click on the **Rg.ServiceCore** project's **References** node and select **Manage NuGet Packages**.
15. In the **NuGet Package Manager** window, select **Browse**.
16. Uncheck **Include prerelease** if it is checked from earlier.
17. Type **StackExchange.Redis** into the search box and press Enter.
18. The **StackExchange.Redis** package should appear. Select it and click **Install**.



19. Repeat the process but this time search for **Newtonsoft.Json**. But DO NOT install the default/latest version.
20. In the list of versions change it to **6.0.4** and then click **Install**.



21. Close the **NuGet Package Manager** window.
22. In **UserOperations.cs**, add these directives to the top of the file:

```
using System.Collections.Generic;
using System.Linq;
using StackExchange.Redis;
using Newtonsoft.Json;
```

23. Add these two fields at the top of the class:

```
private const string RedisConnectionString = "";
private static ConnectionMultiplexer connection =
   ConnectionMultiplexer.Connect(RedisConnectionString);
```

You will fill in the connection string later.

24. Add this nested class inside the **UserOperations** class:

```
        public class CachedUserInfo
        {
            public string Name { get; set; }
            public string Email { get; set; }
            public string AvatarUrl { get; set; }
        }
```

25. In the **RecacheAllUserListAsync** method, add the following code:

```
var allUsersInfo = await dbContext.UserInfos
    .Include(u => u.Avatar)
    .ToListAsync();

IDictionary<string, CachedUserInfo> allUsers = allUsersInfo
    .ToDictionary(
        u => u.Email,
        u => new CachedUserInfo
        {
            Name = u.Name,
            Email = u.Email,
            AvatarUrl = GetAvatarUrl(u)
        });

string serializedUserInfo = JsonConvert.SerializeObject(allUsers);
IDatabase cache = connection.GetDatabase();
await cache.StringSetAsync("AllUserList", serializedUserInfo);
```

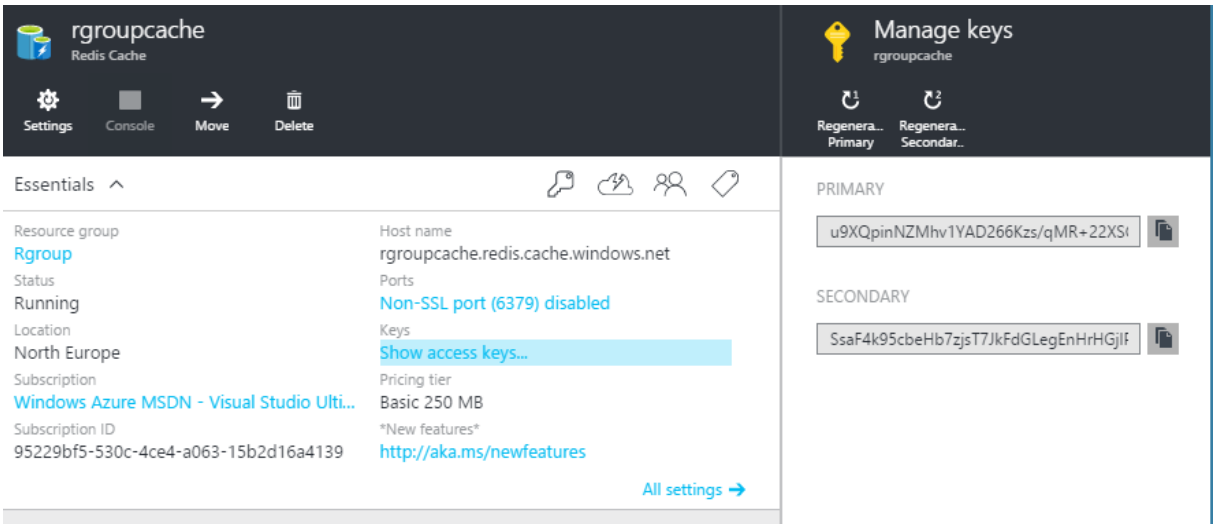
Most of this code just builds the data to be cached. We only interact with the Redis cache right at the very end. It is extremely simple: it acts like a dictionary, storing values by key.

26. Add this method to the class to retrieve the cached data:

```
public static async Task<IDictionary<string, CachedUserInfo>> GetAllUserListAsync(
    ApplicationDbContext dbContext)
{
    IDatabase cache = connection.GetDatabase();
    string cachedData = await cache.StringGetAsync("AllUserList");
    if (cachedData == null)
    {
        await RecacheAllUserListAsync(dbContext);
        cachedData = await cache.StringGetAsync("AllUserList");
    }

    var result = JsonConvert.DeserializeObject<IDictionary<string,
CachedUserInfo>>(cachedData);
    return result;
}
```

27. Go to the Azure portal. It should have finished creating the cache by now. You need to find two pieces of information: the name, which will be at the top of the pane (as well as the first part of **Host name**), and the primary access key, which you can get to by clicking **Show access keys** in the **Essentials** section.



28. Use these to set the contents of the **RedisConnectionString** field you added earlier. The string should take the form:
"CACHENAME.redis.cache.windows.net,ssl=true,abortConnect=false,password=ACCESSKEY"

> Substitute “CACHENAME” and “ACCESSKEY” for your own.

29. Now you can use this to replace the code that looks up the full user list every time. In the **Rg.Web** project’s **Controllers\Ui** folder, open the **UiControllerBase.cs** file.
30. Find its **SetVmEditorInfo** method. This populates the user list on all pages that need it. Replace its contents with this:

```
IDictionary<string, ViewModelWithTextEditingBase.User> allUsers =
    (await UserOperations.GetAllUserListAsync(DbContext)).ToDictionary(
        kv => kv.Key,
        kv => new ViewModelWithTextEditingBase.User
        {
            Name = kv.Value.Name,
            Email = kv.Value.Email,
            AvatarUrl = kv.Value.AvatarUrl
        });
viewModel.AllUsers = allUsers;

return await SetBasicVmInfo(viewModel);
```

31. Put a breakpoint on this method.

> If you removed your password for SQL Server from your web.config, you need to put it back in for testing.

32. Set **Rg.Web** as the start-up project. Run it (**F5**).
33. Log in to your app.
34. When it attempts to show the home page, it will hit your breakpoint. Step into the **GetAllUserListAsync** method. The first time around, the data will not be in the cache. Step through to watch it try and fail to find the data. It will then call **RecacheAllUserListAsync** and try again; it will now succeed. Press **F5** in Visual Studio to let it continue.
35. In the browser, refresh the page. It will hit the breakpoint again. Step through again, and this time note that it gets the data it needs from the cache, avoiding any need to query the database. And it will continue to be able to use the cached data until something triggers a re-cache. Hit **F5** to let the code continue.
36. To verify that the cached data is correct, type "@" in the timeline message textbox. You should see a popup list with all the users in your site (possibly just you) along with their avatar images. This list of usernames and avatars is the reason we need the user list. If this appears, the caching code is working correctly.
37. Close your browser and stop debugging.
38. One last thing: commit your changes and push them up to Visual Studio Team Services.
39. Also, don't forget to mark your Task as done on the Task Board.