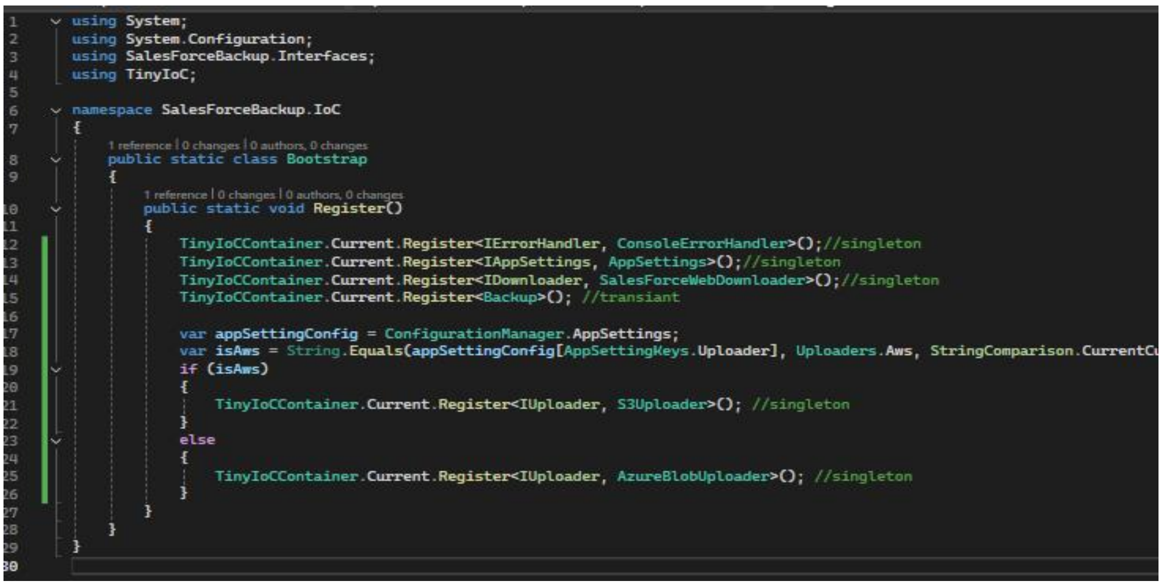
Task Two

1. To avoid Hidden Dependency, I suggest Constructor dependency injection for all of your classes instead of Resolving Dependency by IOC container in the class constructor

   In Bootstrap.cs in Register(), according to AppSettingKeys.Uploader you decided to register S3Uploader or AzureBlobUploader and you instantiated them Manually then to avoiding get stuck to instantiate the other dependency, you decided to resolve the dependency by IOC Container, then drives you to Hidden Dependency as one of the Code Smell. You can write it such as below:



```
using System;
using System.Configuration;
using SalesForceBackup.Interfaces;
using TinyIoC;

namespace SalesForceBackup.IoC
{
    1 reference | 0 changes | 0 authors, 0 changes
    public static class Bootstrap
    {
        1 reference | 0 changes | 0 authors, 0 changes
        public static void Register()
        {
            TinyIoCContainer.Current.Register<IErrorHandler, ConsoleErrorHandler>();//singleton
            TinyIoCContainer.Current.Register<IAppSettings, AppSettings>();//singleton
            TinyIoCContainer.Current.Register<IDownloader, SalesForceWebDownloader>();//singleton
            TinyIoCContainer.Current.Register<Backup>(); //transiant

            var appSettingConfig = ConfigurationManager.AppSettings;
            var isAws = String.Equals(appSettingConfig[AppSettingKeys.Uploader], Uploaders.Aws, StringComparison.CurrentCu
            if (isAws)
            {
                TinyIoCContainer.Current.Register<IUploader, S3Uploader>(); //singleton
            }
            else
            {
                TinyIoCContainer.Current.Register<IUploader, AzureBlobUploader>(); //singleton
            }
        }
    }
}
```

Picture (1)

2. Regarding Single Responsibility you can delegate the responsibility of AssignValuesFromArguments() to AppSettings object because it has already read the setting from ConfigurationManager in ReadAllSettings() then it doesn't need to have the below line codes in Program.cs

   _appSettings = TinyIoCContainer.Current.Resolve<IAppSettings>();
    AssignValuesFromArguments(args);

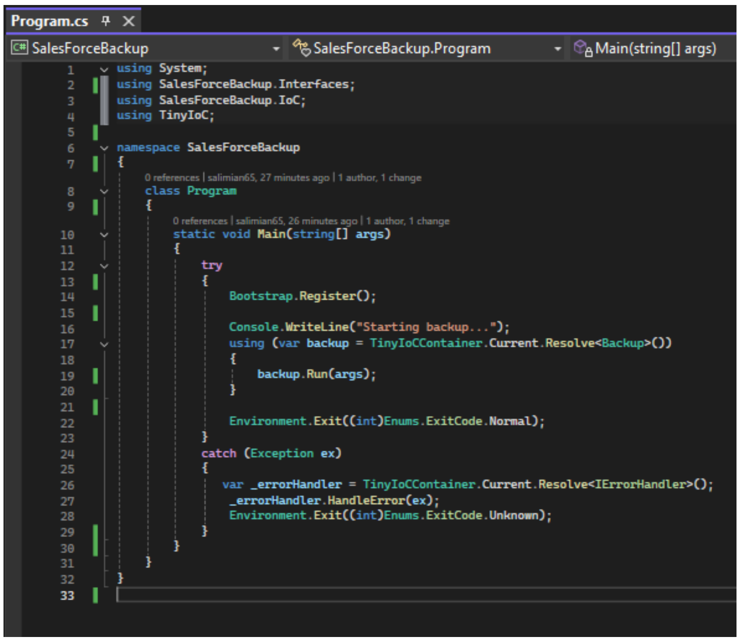
   I prefer Injecting IAppSettings in backup.cs

3. Exception Handling Issue
   First Problem: Environment.Exit(exitCode) in HandleError in ConsoleErrorHandler suppresses and exits the app when gets called, then if you have a finally{} in upper-level or below of catch that calls HandleError(), this finally{} has been never called. Because in catch you exit from the app.

    For example in SalesForceWebDownloader.cs, in the body of Download() we have a try-catch. When we have any exception in Download() the catch in this method, exit the app, then the finally{} in Backup.cs has never had a chance to delete the downloaded file which keeps in _filesToDelete

   Second Problem: redundant try-catch throughout the project that they don't do any specific business.

   Solution: From my point Program.cs is our entry point of application and consequentially is our top-level object. I would handle the general exception at this level.



Program.cs

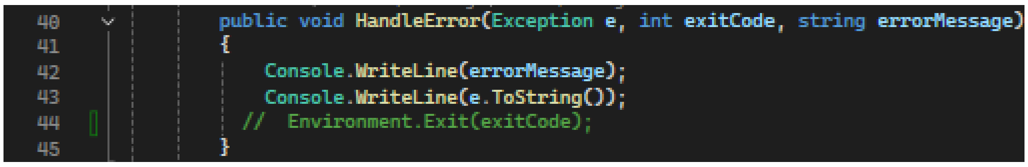
SalesForceBackup / SalesForceBackup.Program / Main(string[] args)

```csharp
using System;
using SalesForceBackup.Interfaces;
using SalesForceBackup.IoC;
using TinyIoC;

namespace SalesForceBackup
{
    0 references | salimian65, 27 minutes ago | 1 author, 1 change
    class Program
    {
        0 references | salimian65, 26 minutes ago | 1 author, 1 change
        static void Main(string[] args)
        {
            try
            {
                Bootstrap.Register();

                Console.WriteLine("Starting backup...");
                using (var backup = TinyIoCContainer.Current.Resolve<Backup>())
                {
                    backup.Run(args);
                }

                Environment.Exit((int)Enums.ExitCode.Normal);
            }
            catch (Exception ex)
            {
              var _errorHandler = TinyIoCContainer.Current.Resolve<IErrorHandler>();
               _errorHandler.HandleError(ex);
               Environment.Exit((int)Enums.ExitCode.Unknown);
            }
        }
    }
}
```

Picture (2)

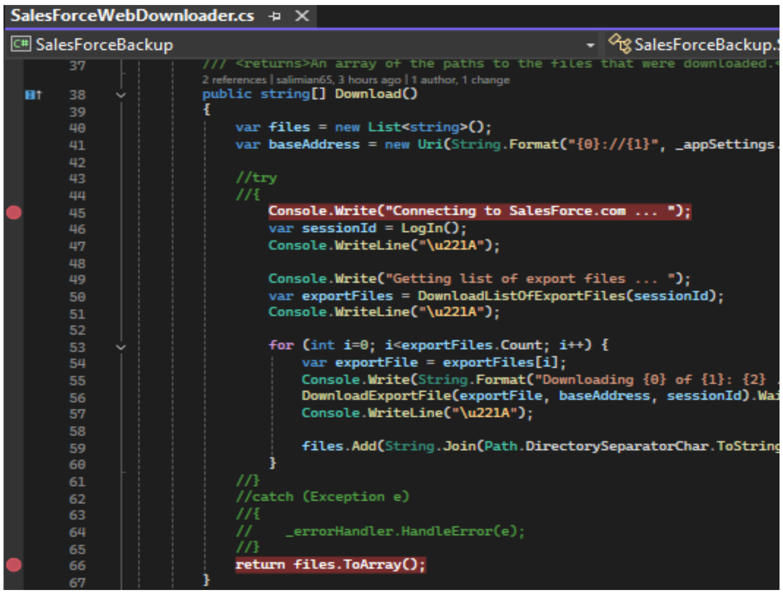
To reach this purpose, we should make some changes:

First: we don't need Environment.Exit(exitCode); in HandleError in ConsoleErrorHandler



```csharp
public void HandleError(Exception e, int exitCode, string errorMessage)
{
    Console.WriteLine(errorMessage);
    Console.WriteLine(e.ToString());
 //  Environment.Exit(exitCode);
}
```

Second: Now we can delete all the general try catch in other class. Because we catch all of them in Main() of Programs.cs

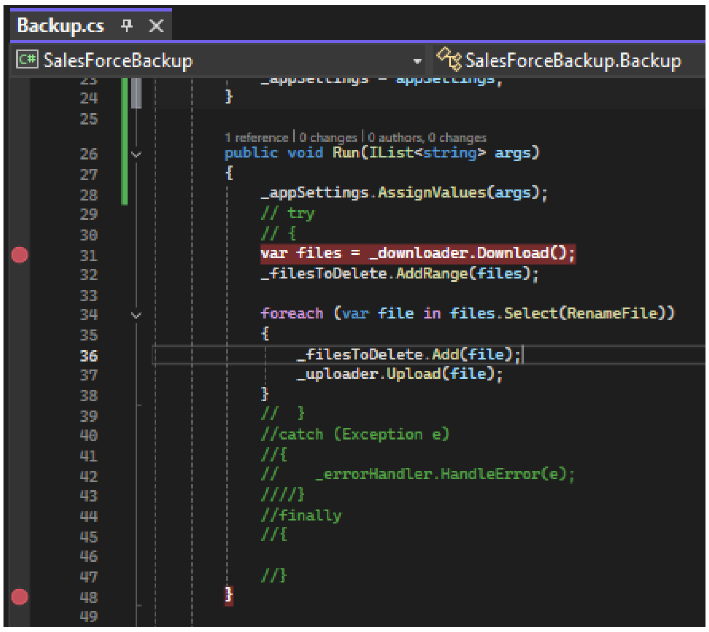


SalesForceWebDownloader.cs

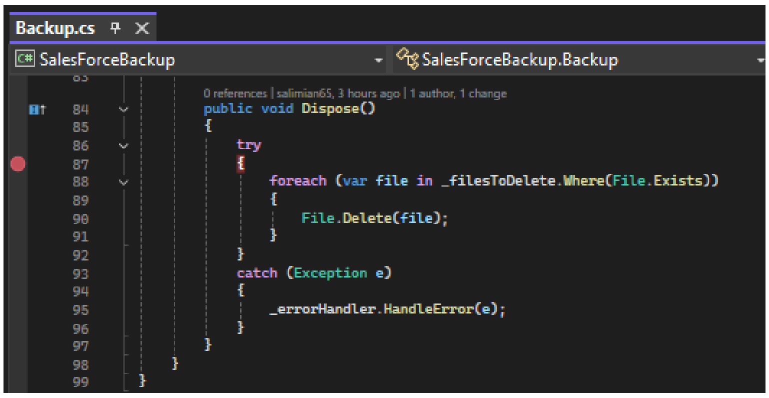
SalesForceBackup / SalesForceBackup.

```csharp
        /// <returns>An array of the paths to the files that were downloaded.
        2 references | salimian65, 3 hours ago | 1 author, 1 change
        public string[] Download()
        {
            var files = new List<string>();
            var baseAddress = new Uri(String.Format("{0}://{1}", _appSettings.

            //try
            //{
                Console.Write("Connecting to SalesForce.com ... ");
                var sessionId = LogIn();
                Console.WriteLine("\u221A");

                Console.Write("Getting list of export files ... ");
                var exportFiles = DownloadListOfExportFiles(sessionId);
                Console.WriteLine("\u221A");

                for (int i=0; i<exportFiles.Count; i++) {
                    var exportFile = exportFiles[i];
                    Console.Write(String.Format("Downloading {0} of {1}: {2}
                    DownloadExportFile(exportFile, baseAddress, sessionId).Wa
                    Console.WriteLine("\u221A");

                    files.Add(String.Join(Path.DirectorySeparatorChar.ToString
                }
            //}
            //catch (Exception e)
            //{
            //    _errorHandler.HandleError(e);
            //}
            return files.ToArray();
        }
```

4. Backup.cs class can implement IDisposable and in Despose() method check the business of File deletion. Consequentially we can delete Try Catch Finally in Run() from this file.



```
        _appSettings = appSettings;
    }

    1 reference | 0 changes | 0 authors, 0 changes
    public void Run(IList<string> args)
    {
        _appSettings.AssignValues(args);
        // try
        // {
        var files = _downloader.Download();
        _filesToDelete.AddRange(files);

        foreach (var file in files.Select(RenameFile))
        {
            _filesToDelete.Add(file);
            _uploader.Upload(file);
        }
        //  }
        //catch (Exception e)
        //{
        //    _errorHandler.HandleError(e);
        ////}
        //finally
        //{

        //}
    }
```

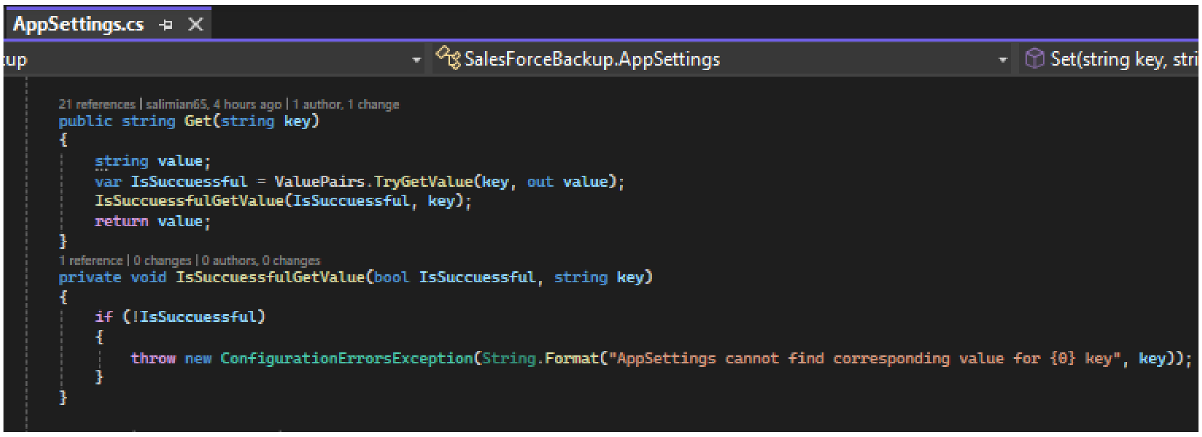


```
        0 references | salimian65, 3 hours ago | 1 author, 1 change
        public void Dispose()
        {
            try
            {
                foreach (var file in _filesToDelete.Where(File.Exists))
                {
                    File.Delete(file);
                }
            }
            catch (Exception e)
            {
                _errorHandler.HandleError(e);
            }
        }
    }
}
```

Also, we should call backup.run() in using (){} in Program.cs. that is guaranteed Dispose() gets called for releasing the resource and memory as you can see in the picture (2) line 17

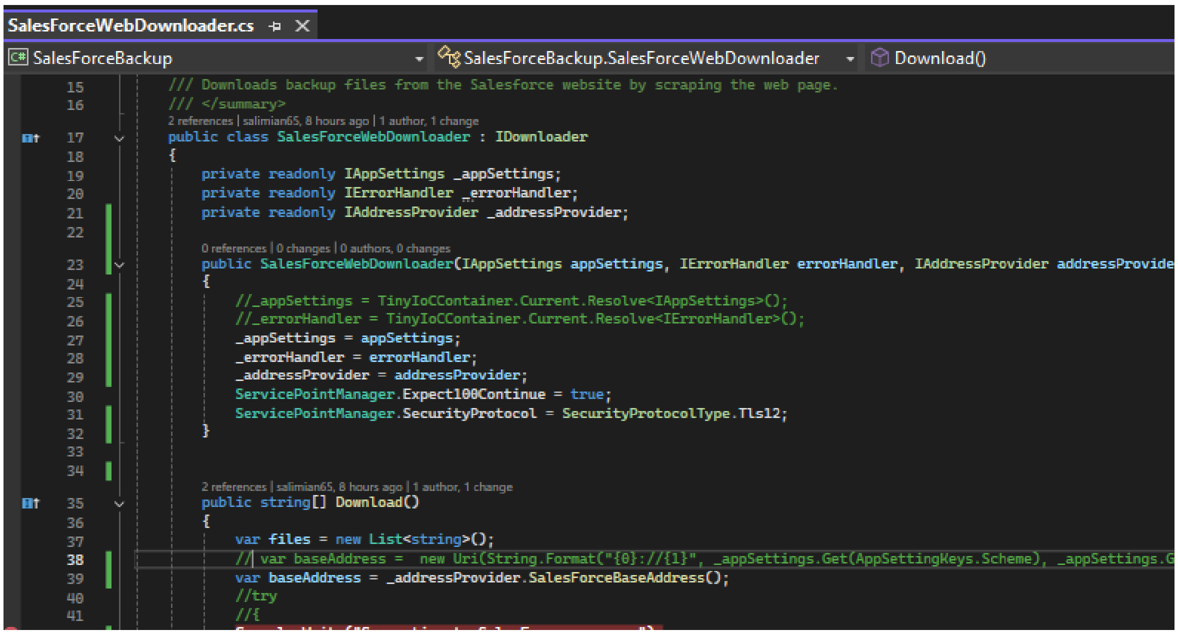
5. It doesn't need to keep AppSettings properties static, if your purpose was that keep them alive during the app life-cycle, you have already registered AppSettings as a singleton instance. From my perspective, I stay away from static classes or methods as much as possible unless in a rare situation.

6. Get(string key) in AppSettings need to check if ValuePairs.TryGetValue(key, out value); cannot find the key. Then we can have a guard that if there isn't a key throw an exception.



```
    21 references | salimian65, 4 hours ago | 1 author, 1 change
    public string Get(string key)
    {
        string value;
        var IsSuccuessful = ValuePairs.TryGetValue(key, out value);
        IsSuccessfulGetValue(IsSuccuessful, key);
        return value;
    }
    1 reference | 0 changes | 0 authors, 0 changes
    private void IsSuccuessfulGetValue(bool IsSuccuessful, string key)
    {
        if (!IsSuccuessful)
        {
            throw new ConfigurationErrorsException(String.Format("AppSettings cannot find corresponding value for {0} key", key));
        }
    }
```

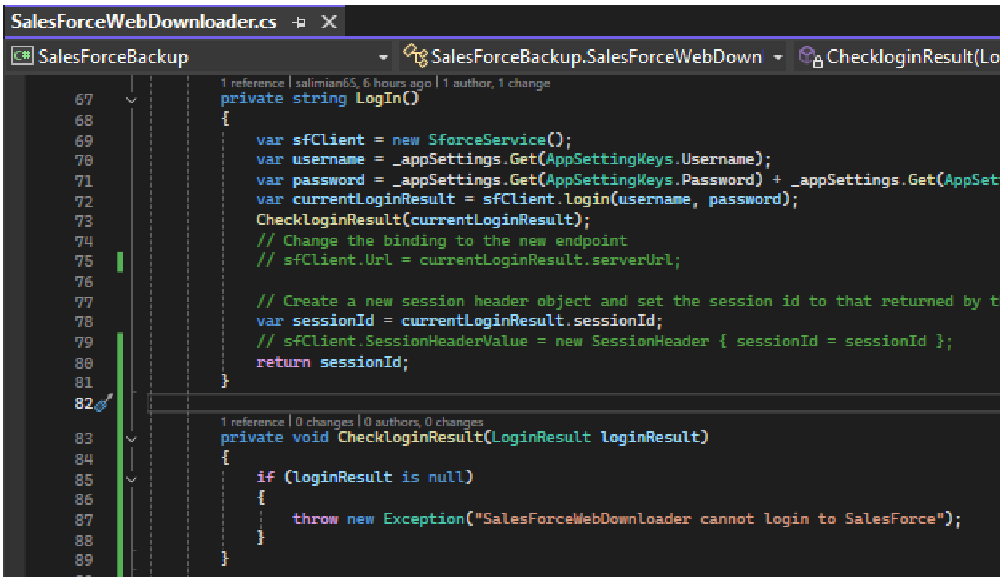
7. In Download() method in SalesForceWebDownloader the baseAddress is built by some format and setting. And it isn't only here, there are a lot of places where we create some URLs and addresses for reaching our third-party provider. According to single responsibility, I prefer having another essence and nature that can be considered as a service (Pure Fabrication) to provide us with addresses and call it AddressProvider.


SalesForceWebDownloader.cs
SalesForceBackup | SalesForceBackup.SalesForceWebDownloader | Download()

```
15      /// Downloads backup files from the Salesforce website by scraping the web page.
16      /// </summary>
        2 references | salimian65, 8 hours ago | 1 author, 1 change
17      public class SalesForceWebDownloader : IDownloader
18      {
19          private readonly IAppSettings _appSettings;
20          private readonly IErrorHandler _errorHandler;
21          private readonly IAddressProvider _addressProvider;
22
            0 references | 0 changes | 0 authors, 0 changes
23          public SalesForceWebDownloader(IAppSettings appSettings, IErrorHandler errorHandler, IAddressProvider addressProvide
24          {
25              //_appSettings = TinyIoCContainer.Current.Resolve<IAppSettings>();
26              //_errorHandler = TinyIoCContainer.Current.Resolve<IErrorHandler>();
27              _appSettings = appSettings;
28              _errorHandler = errorHandler;
29              _addressProvider = addressProvider;
30              ServicePointManager.Expect100Continue = true;
31              ServicePointManager.SecurityProtocol = SecurityProtocolType.Tls12;
32          }
33
34
            2 references | salimian65, 8 hours ago | 1 author, 1 change
35          public string[] Download()
36          {
37              var files = new List<string>();
38              // var baseAddress =  new Uri(String.Format("{0}://{1}", _appSettings.Get(AppSettingKeys.Scheme), _appSettings.G
39              var baseAddress = _addressProvider.SalesForceBaseAddress();
40              //try
41              //{
```

As you can see in the above picture _addressProvider is injected and in line 39 _addressProvider.SalesForceBaseAddress() is substituted instead of line 38. We can expand it throughout the project which brings us more readability, maintainability, reusability entire the project and when we change the address or format we need to change just one place, also for testability we can mock it or fake it or use any other test double

8. In LogIn() method in SalesForceWebDownloader if the login fails we should have a guard and handle it with the appropriate exception


SalesForceWebDownloader.cs
SalesForceBackup | SalesForceBackup.SalesForceWebDown | CheckloginResult(Lo

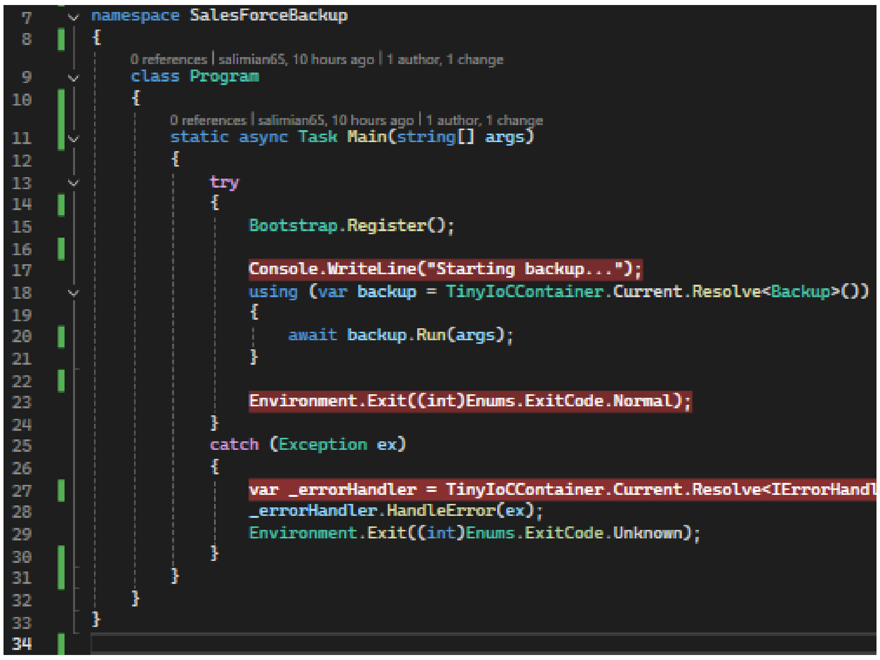
```
        1 reference | salimian65, 6 hours ago | 1 author, 1 change
67      private string LogIn()
68      {
69          var sfClient = new SforceService();
70          var username = _appSettings.Get(AppSettingKeys.Username);
71          var password = _appSettings.Get(AppSettingKeys.Password) + _appSettings.Get(AppSet
72          var currentLoginResult = sfClient.login(username, password);
73          CheckloginResult(currentLoginResult);
74          // Change the binding to the new endpoint
75          // sfClient.Url = currentLoginResult.serverUrl;
76
77          // Create a new session header object and set the session id to that returned by t
78          var sessionId = currentLoginResult.sessionId;
79          // sfClient.SessionHeaderValue = new SessionHeader { sessionId = sessionId };
80          return sessionId;
81      }
82
        1 reference | 0 changes | 0 authors, 0 changes
83      private void CheckloginResult(LoginResult loginResult)
84      {
85          if (loginResult is null)
86          {
87              throw new Exception("SalesForceWebDownloader cannot login to SalesForce");
88          }
89      }
```

The other point here is lines 75 and 79, why sfClient.Url and sfClient.SessionHeaderValue is initialized, while only sessionId is returned. The scope of sfClient instance in line 69 is just during the login method and it doesn't have any effect. Then I would delete lines 75, and 79.

9. `DownloadExportFile`() method in SalesForceWebDownloader doesn't need to be a static method

10. Why DownloadWebpage() method in SalesForceWebDownloader isn't an async Method? Instead of taking the Result of client.SendAsync() we can await it, Await is an asynchronous wait but the result is a blocking wait.

Note: As you know we have a console app which seems for each time it gets run and take a backup then get closed at the end automatically. current code shows we don't have business requirements for parallel processing. Then asynchronous wait may not have any place for discussion. But I prefer to have a comprehensive approach and design through entire the app. For example instead of having DownloadExportFile() method as async and DownloadWebpage() method as sync, I prefer having both of them as Async (one signature), then finally await backup.run() in the Main method in Progaram.cs.

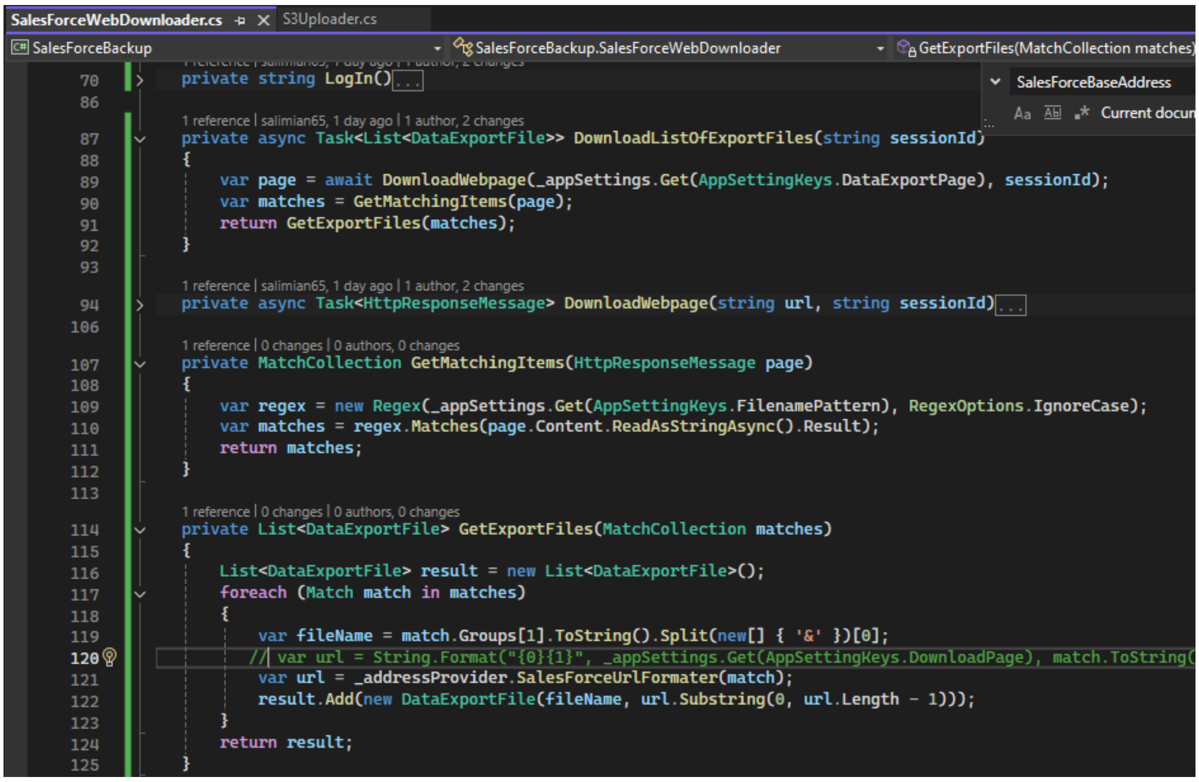


```
namespace SalesForceBackup
{
    0 references | salimian65, 10 hours ago | 1 author, 1 change
    class Program
    {
        0 references | salimian65, 10 hours ago | 1 author, 1 change
        static async Task Main(string[] args)
        {
            try
            {
                Bootstrap.Register();

                Console.WriteLine("Starting backup...");
                using (var backup = TinyIoCContainer.Current.Resolve<Backup>())
                {
                    await backup.Run(args);
                }

                Environment.Exit((int)Enums.ExitCode.Normal);
            }
            catch (Exception ex)
            {
                var _errorHandler = TinyIoCContainer.Current.Resolve<IErrorHandl
                _errorHandler.HandleError(ex);
                Environment.Exit((int)Enums.ExitCode.Unknown);
            }
        }
    }
}
```

11. Download() method in SalesForceWebDownloader instead of returning files.ToArray() can return files; consequentially in the method return signature instead of string[] would be better return Task<List<string>>.

12. I extract two additional functionalities which can separate them into another method in DownloadListOfExportFiles() method in SalesForceWebDownloader

    var page = await DownloadWebpage(_appSettings.Get(AppSettingKeys.DataExportPage), sessionId);
    var matches = GetMatchingItems(page);
    return GetExportFiles(matches);

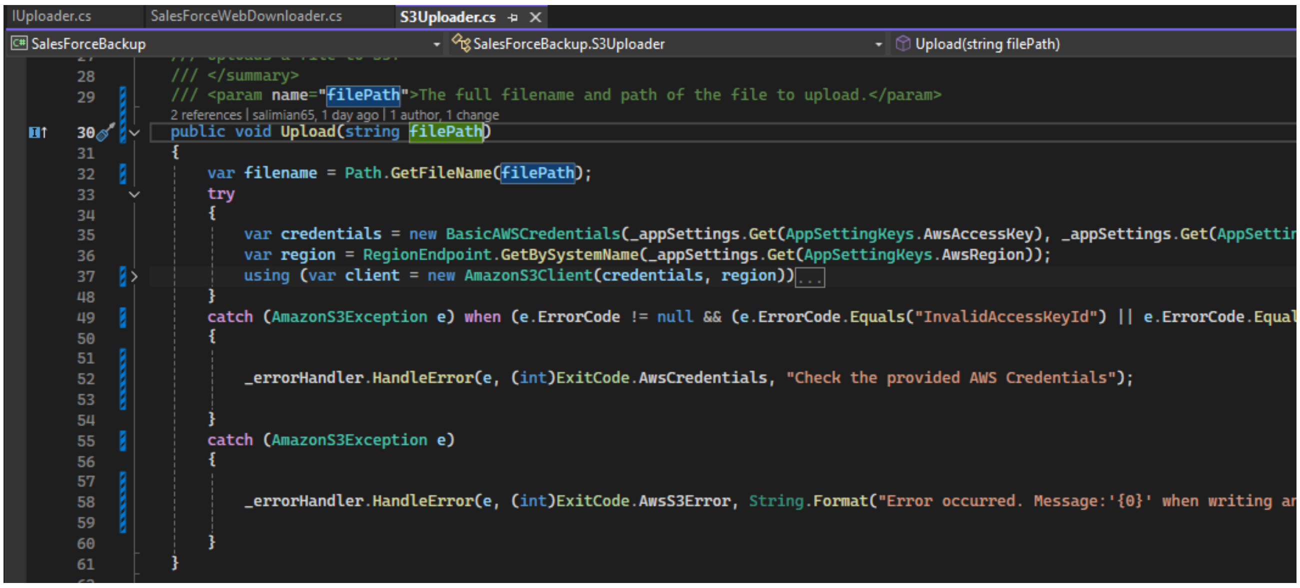


SalesForceWebDownloader.cs | S3Uploader.cs
SalesForceBackup | SalesForceBackup.SalesForceWebDownloader | GetExportFiles(MatchCollection matches)

```
private string LogIn()...

1 reference | salimian65, 1 day ago | 1 author, 2 changes
private async Task<List<DataExportFile>> DownloadListOfExportFiles(string sessionId)
{
    var page = await DownloadWebpage(_appSettings.Get(AppSettingKeys.DataExportPage), sessionId);
    var matches = GetMatchingItems(page);
    return GetExportFiles(matches);
}

1 reference | salimian65, 1 day ago | 1 author, 2 changes
private async Task<HttpResponseMessage> DownloadWebpage(string url, string sessionId)...

1 reference | 0 changes | 0 authors, 0 changes
private MatchCollection GetMatchingItems(HttpResponseMessage page)
{
    var regex = new Regex(_appSettings.Get(AppSettingKeys.FilenamePattern), RegexOptions.IgnoreCase);
    var matches = regex.Matches(page.Content.ReadAsStringAsync().Result);
    return matches;
}

1 reference | 0 changes | 0 authors, 0 changes
private List<DataExportFile> GetExportFiles(MatchCollection matches)
{
    List<DataExportFile> result = new List<DataExportFile>();
    foreach (Match match in matches)
    {
        var fileName = match.Groups[1].ToString().Split(new[] { '&' })[0];
       // var url = String.Format("{0}{1}", _appSettings.Get(AppSettingKeys.DownloadPage), match.ToString(
        var url = _addressProvider.SalesForceUrlFormater(match);
        result.Add(new DataExportFile(fileName, url.Substring(0, url.Length - 1)));
    }
    return result;
}
```

13. In IUploader interface I would name the input of Upload as a filePath instead of file. Because the nature of it is path

14. In S3Uploader we have a try-catch and for better performance, I would use when Syntax in front of catch instead of check the if in Catch

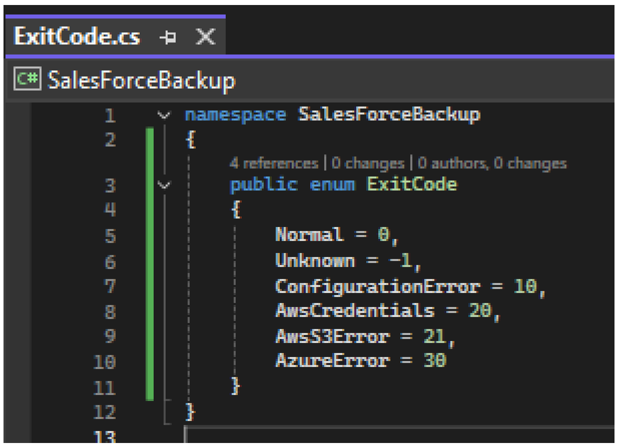


IUploader.cs | SalesForceWebDownloader.cs | S3Uploader.cs

SalesForceBackup | SalesForceBackup.S3Uploader | Upload(string filePath)

```
/// </summary>
/// <param name="filePath">The full filename and path of the file to upload.</param>
2 references | salimian65, 1 day ago | 1 author, 1 change
public void Upload(string filePath)
{
    var filename = Path.GetFileName(filePath);
    try
    {
        var credentials = new BasicAWSCredentials(_appSettings.Get(AppSettingKeys.AwsAccessKey), _appSettings.Get(AppSettin
        var region = RegionEndpoint.GetBySystemName(_appSettings.Get(AppSettingKeys.AwsRegion));
        using (var client = new AmazonS3Client(credentials, region))...
    }
    catch (AmazonS3Exception e) when (e.ErrorCode != null && (e.ErrorCode.Equals("InvalidAccessKeyId") || e.ErrorCode.Equal
    {

        _errorHandler.HandleError(e, (int)ExitCode.AwsCredentials, "Check the provided AWS Credentials");

    }
    catch (AmazonS3Exception e)
    {

        _errorHandler.HandleError(e, (int)ExitCode.AwsS3Error, String.Format("Error occurred. Message:'{0}' when writing ar

    }
}
```

15. It doesn't need to keep the ExitCode enum inside Enums class. You can take ExitCode enum out of this class and delete the Enums class. Also, edit the file name from Enums.cs to Exitcode.cs



ExitCode.cs

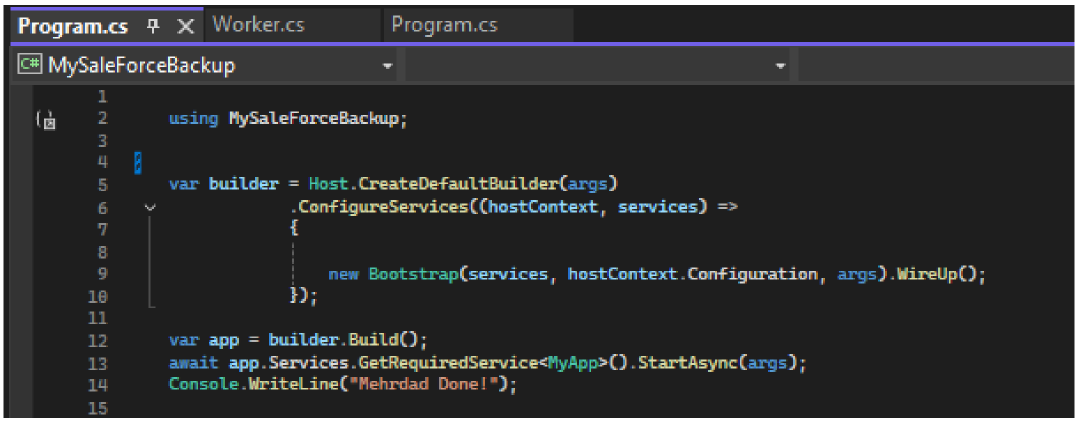
SalesForceBackup

```
namespace SalesForceBackup
{
    4 references | 0 changes | 0 authors, 0 changes
    public enum ExitCode
    {
        Normal = 0,
        Unknown = -1,
        ConfigurationError = 10,
        AwsCredentials = 20,
        AwsS3Error = 21,
        AzureError = 30
    }
}
```

//

16. Up to now we reviewed the code as it is, for avoiding off-topic discussion I assumed this code is base on our requirement and just reviewed the code. Maybe our servers are only able to execute the .net framework, not .net 6,7,8. But from now on as additional discussion, I want to take the step forward and discuss beyond the present topic.
    1) The code should have a test coverage. We consider the test as the first client which tests the business. We consider the test as an online document.
    2) If the commented explanation above each method or class is part of our team convention, I don't have a problem with it, but if not, I prefer not to have them, instead, I use self-explanatory naming for each element also using unit test coverage. Then my code review start point is reviewing the test scenario to understand the functionality.
    3) Now is September of 2024, if we assume that we don't have any issues with using .net core or .net 7,8, then I prefer having a .net 8 one, it is a small project and it doesn't take much effort to migrate it to .net 8.
        a. Note: My-dotNet8-Refactored-dSalesForceBackup project is accessible in my GitHub by this address . it is WIP (work in progress) and I am working on it now.
    4) We assume knowing about the advantages of .net 8.

5) We can Use the advantages of IHostBuilder also can have a BackgroundService which named it as a worker. The worker can listen to the channel or TPL Dataflow.



```
using MySaleForceBackup;


var builder = Host.CreateDefaultBuilder(args)
                .ConfigureServices((hostContext, services) =>
                {

                    new Bootstrap(services, hostContext.Configuration, args).WireUp();
                });

var app = builder.Build();
await app.Services.GetRequiredService<MyApp>().StartAsync(args);
Console.WriteLine("Mehrdad Done!");
```

6) Then we can design entire of the flow as an async process. User from the UI send a request for making a new backup, then as soon as request comes to the Controller, we dispatch BackupCommand to download the files form SalesForceWebPage and put them on the channel, then respond to customer that “the backup is processing”. Also, we can use SignalR as a push notification to show a progress bar to customer. Then Worker is listening to channel and take the downloaded file one by one from the channel and try to process them and uploads them as an Uploader to Azure or AWS. When the process is finished then push a notification to the customer that it gets finished. I will elaborate it in Third Task.

7) Using HttpClientFactory instead of HttpClient for taking the advantage of httpClientPool and having better management on http request

Note: if you see any commented code in above picture or in my code in GitHub, I only keep them for showing the changes that I have. Otherwise, all of them should be deleted in Basecode