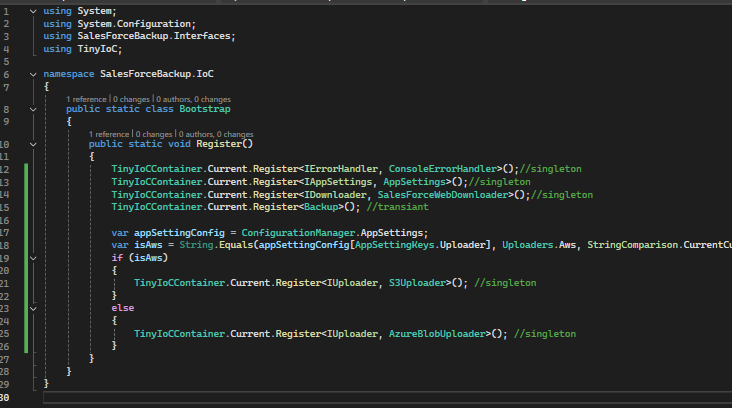
1. To avoid Hidden Dependency, I suggest Constructor dependency injection for all of your class instead of Resolving Dependency by IOC container in class constructor

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



Picture (1)

1. Regarding Single Responsibility you can delegate the responsibility of AssignValuesFromArguments() to AppSettings object because it has already read 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

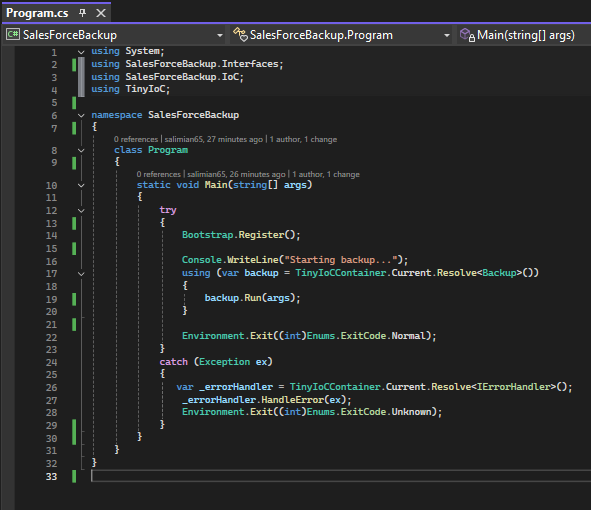
1. Exception Handling Issue

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

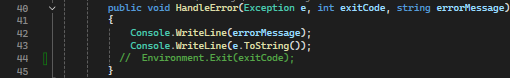
Solution: from my point Program.cs is our entry point of application and consequentially is our top-level object. I would handle the general exception in this level.



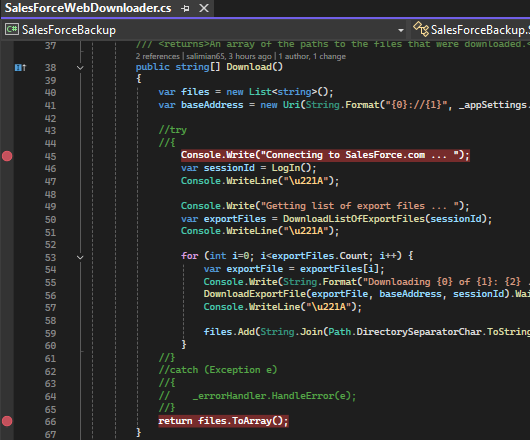
Picture (2)

For reaching to this purpose, we should do some changes:

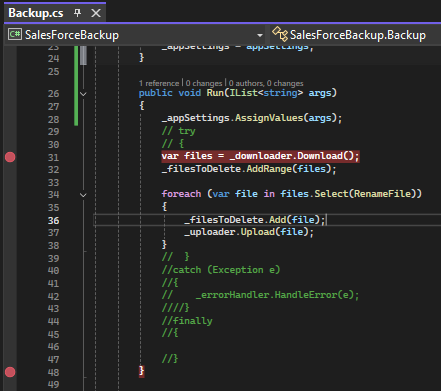
First: we don’t need Environment.Exit(exitCode); in HandleError in ConsoleErrorHandler

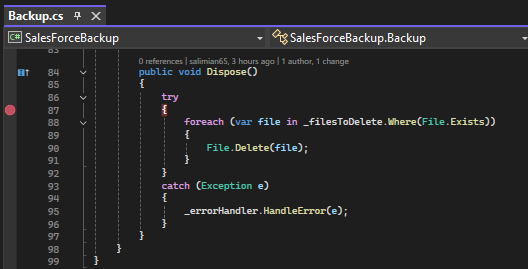


Second: Now we can delete all the general try catch in other class. Because we catch all of them in Main() of Programs.cs



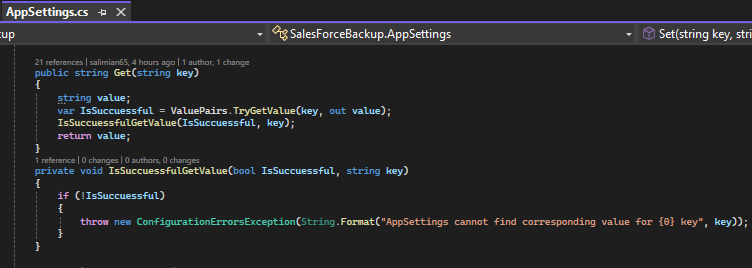
1. Backup.cs class can implement IDisposable and in Despose() method check the business of File deletion. Consequentially we can delete Try Catch Finally in Run() from this file.



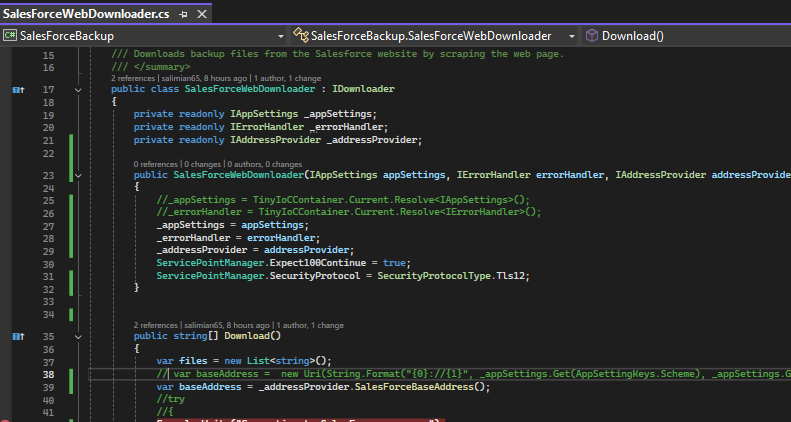


Also we should call backup.run() in using (){} in Program.cs. that is garanteed Dispose() get called for releasing the resource and memory as you can see in picture (2) line 17

1. It doesn’t need to keep AppSettings properties static, if your purpose was that keeping them alive during the app life-cycle, you have already register 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.
2. 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.

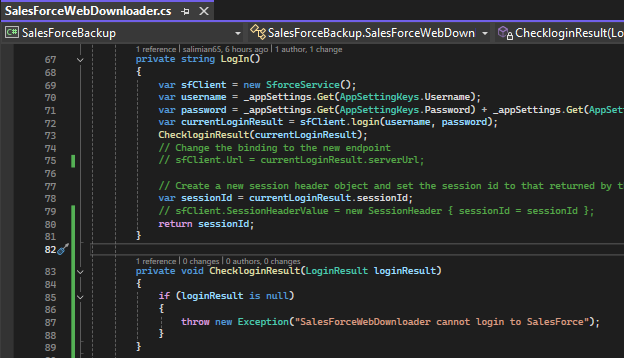


1. In Download() method in SalesForceWebDownloader the baseAddress is built by some format and setting. And it isn’t only here, there are a lot of places we create some URLs and address for reaching to our third-party provider. According to single responsibility I prefer having another essence and nature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



As you can see in above picture \_addressProvider is injected and in line 39 \_addressProvider.SalesForceBaseAddress() is substituted instead of line 38. We can expand it throughout the project that 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 using any other test double

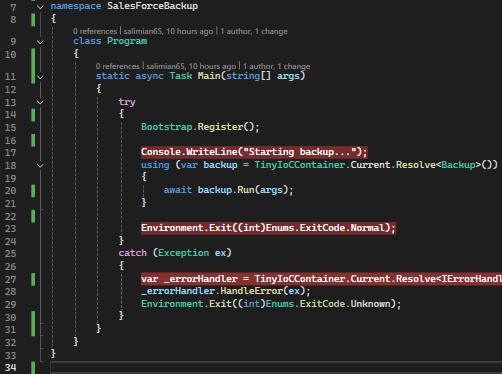
1. In LogIn() method in SalesForceWebDownloader if the login get fail we should have a guard and handle it with appropriate exception



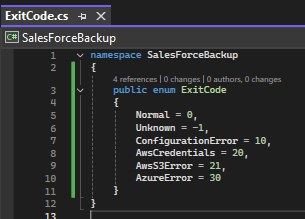
The other point here is line 75 and 79, why sfClient.Url and sfClient.SessionHeaderValue are initialized, while only sessionId is returned. The scope of sfClient instance in line 69 is just during the login method and it doesn’t have any effect. Then I would delete lines 75, 79.

1. DownloadExportFile() method in SalesForceWebDownloader doesn’t need to be static method
2. Why DownloadWebpage() method in SalesForceWebDownloader isn’t an async Method? Instead of take the Result of client.SendAsync() we can await it, Await is an asynchronous wait but 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 requirement for parallel processing. Then asynchronous wait maybe doesn’t have any place for discussion. But I prefer 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 Main method in Progaram.cs.



1. It doesn’t need to keep 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



1. I don’t have a problem using TinyIoCContainer until at least it provides our minimum and basic requirements.

If we consider the scenario that we have many projects in our team then I prefer using a comprehensive tools and solution entire projects in our team. Choose appropriate IOC container is one of the design decision which we can have in our team, there are a lot of mature, and simple IOC in the market that have a good community, Contribution and support some feature