Code Test

We would like you to refactor the GetLearner method in the LearnerService class. When refactoring you should consider the following; SOLID principles, maintainability, testing.

You can use any framework(s) of your choice. We are not expecting the work to be finished, we will expect you to discuss your approach and any further improvement you would make.

You can make any changes, apart from the following

* Signature of the ArchivedDataService -> returns Learner
* Signature of the LearnerDataAccess -> returns LearnerResponse
* Signature of the FailoverLearnerDataAccess-> returns LearnerResponse

The GetLearner method is responsible for executing the following logic

* Based on the isLearnerArchived parameter retrieving Learners from the archive
* The main Learner data store is a 3rd party service (which doesn’t have particularly high SLA), so therefore a failover data store has been created which stores a backup copy of the Learner records
* The method evaluates if the system should be in failover mode based on a given number of failed requests in a given time period (currently 10 minutes)
* If the system is in failover mode Learners are retrieved from the failover store

The refactored solution must compile and should be accompanied by Unit Tests

The use of the Decorator Pattern satisfies the following OO coding principles.

SOLID

Single Responsibility

The only reason to change the ILearnerReadService interface is when changing how an instance of Learner is being retrieved.

Open Closed Principle

Any number of classes can implement the ILearnerReadService interface without the need to change the ILearnerReadService interface.

Lyscov Substitution Principle

Any class implementing the ILearnerReadService interface can be substituted where the ILearnerReadService interface is used.

Dependency Inversion Principle

None of the data access classes are referenced within the Business Logic Façade, only the ILearnerReadService interface is exposed.

Interface Segregation Principle

Segregating the comprehensive Learner Service down into Read, Write and Delete makes it more likely that each required implementation does not unnecessarily have to implement methods which aren’t required of its specification. i.e. The use of such constraints as throw NotImplementedException in unused methods is reduced.

DataAccess,Logic,Model is just preliminary without knowledge of where it fits amongst other code but these folders would hopefully eventually be their own VS projects.

Model entities are shared here but most likely Business Entities will be mapped from Data Transfer Objects (DTO’s) and/or UI Models.

Split out interfaces into Interface only project.

Archived data is separate to current live data, failover data is not archived data.

Split Code into Logic, Model & DataAccess areas to identify and segregate common functionality and reduce dependencies as much as possible.

Why is FailoverLearnerDataAccess.GetLearnerById a static method?

removing dependency on System.Configuration

Requires a code coverage tool of some description, VS Enterprise has it built in but I’m not using that currently.