.NET Assessment #1 - SmartShare

Assessment Overview

For this assessment, you will be creating a command-line interfaceapplication for 'securely' storing files. Files will be uploaded with a 'password'. If another user wants to download the file, they must provide the file name and password in order to download it. The application will be split into two parts - the client and the server. It will have support for concurrency using the the .NET concurrency/asynchronous API and will store files and related information in a relational database.

Command-line applications

The client side of this application is a command-line application. Command-line applications, also referred to as console applications, are computer programs designed to be used from a text interface, such as a shell. Command-line applications usually accept various inputs as arguments, often referred to as parameters or sub-commands, as well as options, often referred to as flags or switches. Some popular and widely-used command-line applications include grep, git, and curl.

When creating any software that a user will be directly interacting with, it is important to make sure that the 'learning curve' is as smooth and as low as reasonably possible. Making things unique, over-complicated, or going against the grain of the majority of other applications is a great way to ensure that no one will want to use your software. This is especially important for command-line applications. The world of command-line applications is one with a rich history, extensive refinement, and tried and tested standards. Because of this, it is important to follow convention and utilize (as much as possible) what is generally considered best-practices for building CLI applications.

Creating command-line applications is difficult. Parsing text, arguments, options, and providing feedback to the user are just a few of the things you need to consider when writing an easy-to-use command-line application. Like anything in programming, we don't want to re-invent the wheel. We will be using **CommandLineParser** for our client-side .NET CLI application. It provides an easy-to-use API with minimal boilerplate. Their documentation is sparse and unorganized, but should be referenced first for any issues related to the CLI portion of this assessment..

Requirements Overview

Concurrency

When a client connects to the server, the server should create a new `Thread` *or* `Task` *to handle the request. The thread/task should run until the response is given to the client.*

Uploading

When the user wants to upload a file, they may provide a 'relative path' to the file or an 'absolute path'. They have the option of providing their own password which will act as the 'key' or 'password'

for people wishing to download the file. If they do not provide their own password, a password will be generated for them. This functionality is provided for you in the skeleton of the assessment.

File names should be unique. If a file is uploaded with the same name as another file, it should fail to upload.

Additional features for uploading

- **Expiration** - The client should be able to provide an optional flag denoting the number of minutes until the file expires and can no longer be accessed. If a value is not provided, the expiration should default to `60 minutes` from the time the file was uploaded. The maximum value allowed should be `1440` minutes (24 hours) and the minimum value allowed should be `1` minute.
- **Maximum Downloads** - The client should be able to provide an optional flag denoting the maximum number of downloads available until the file can no longer be accessed. If a value is not provided, there should be no restriction on the number of downloads available.

Downloading

When a user wants to retrieve a file, they will be required to provide a filename and a password.

There are four cases in which the user should not be able to retrieve the specified file.

- The file doesn't exist
- The password doesn't match the password provided when the file was uploaded
- The file has 'expired' - the time has passed that was specified when the file was uploaded
- The maximum downloads have been reached - this value was passed when the file was uploaded

If a download fails, the user should be displayed with *the same* error message **in all cases.**

It is up to you where the files are stored once they're downloaded. It is recommended to store them relative to where the application was executed. If run from within visual studio, this will likely be the root of the `Client`project.

File Deletion

If a file is expired or the maximum downloads have been reached, the file entry in the database should be deleted.

Viewing

When a user wants to view a file's current 'summary', they must provide the file name and a correct password. They should then be shown the time created, remaining downloads and time until expiration.

Provided Skeleton

Disclaimer: *The skeleton provided is a working example created for the purpose of showing **one** way to approach this problem. It is written to be enough scaffolding to build on top of as well as demonstrate how to use CommandLineParser. However, it is entirely up to you exactly how the user will interact with the CLI as long as the requirements listed below are met. In the case that you are satisfied with the skeleton's implementation, feel free to continue building on top of it.*

The skeleton contains a basic boilerplate showcasing some of the basic features of CommandLineParser. You are given a skeleton that provides a working command-line interface which has pre-written 'download' and 'upload' commands and their arguments and responds by printing verification messages to the console. One of the features that it utilizes is**verbs**. This enables the ability to delineate and separate options and values for multiple commands within a single application. The skeleton also contains an incomplete `Api` class in the `Api` directory and an empty `SmartShareServer` class in the `Server` project. The skeleton does not contain any code that connects the client to the server. The server side of the skeleton is left mostly empty and it is up to you to create a server which supports simultaneous requests as well as provides storage of files in a database. A `SmartShareContext` class has been provided with a connection string, but lacks the context definition and models needed to use Entity Framework Core in order to interact with the database.

You are also given a `schema.sql` which you can use to generate an appropriate schema and tables to be used when creating your `Entity`classes and context.

Running/Testing the Client Application

In order to test and run your application, it is recommended to run commands programatically. Examples of this are inside of `Client/Program.cs`. This main method is set to run commandline arguments through a simulated `__YourMain` method. The usage of these should be self-explanatory, but if there is any confusion, please ask an instructor for clarification.

Bonus Challenge

*This bonus challenge should **only** be attempted after completion of all of the above requirements. If any of the above requirements are not met, and the bonus challenge is attempted, you will not receive any extra credit and you will be deducted additional points for attempting extra work without meeting the base requirements.*

Modify the schema and application to support uploading/downloading multiple files under a single request. The files should be uploaded with a single password and retrieved with the same password. All other upload/download requirements still apply.

Criteria	Grading Scale		
Criteria	**Grading Scale**		
Concurrency Server creates threads/tasks for each incoming client connection	**15** Completed	**10** Mostly completed but may have bugs	**5** Barely functioning - Shows lack of understanding of how to use concurrency/tasks in .NET/C#
Entity Framework Core Created appropriate entities and context for interacting with the database through the ORM	**15** Completed	**10** Mostly completed but may have bugs	**5** Barely functioning - Shows lack of understanding of how ORM/EF Core works or how to interact with a database using entities/context
Upload User is able to upload a file via a command provided with a password	**15** Completed according to specification	**10** Mostly completed but may have bugs	
Download User is able to download a file via a command provided the correct password is given	**15** Completed according to specification	**10** Mostly completed but may have bugs	
View/Summary User is able to view a summary via a command provided of a file's current state, provided the correct password is given, and be displayed with the information specified in the requirements.	**10** Completed according to specification		
Expiration User can provide an expiration in minutes. Once the expiration time is reached, the file can no longer be downloaded	**10** Completed	**5** Mostly completed but may have bugs or not match specification	
Max Downloads User is able to provide a maximum number of downloads available for their uploaded file. Once that number of downloads has been reached, the file entry is deleted from the database.	**10** Completed	**5** Mostly completed but may have bugs or not match specification	
Code Quality Code Smell (Formatting) Code Smell (Leftover Code/Comments) Code Smell (Lack of Abstraction/Reuse) Code Smell (Lack of Library Use/Understanding/Research) Best Practices (Language) Best Practices (Library/Framework) Best Practices (Topic) Design (Coding Patterns) Design (Organizational Patterns)	**10** Completed		

Criteria	Grading Scale
Bonus Challenge If student attempts this but has bugs or other parts of the assessment that are incomplete, they receive zero credit for the bonus challenge and additional points should be taken off. Modify the schema and application to support uploading/downloading multiple files under a single request. The files should be uploaded with a single password and retrieved with the same password. All other upload/download requirements still apply.	**10** Completed