İSTANBUL
ÜNİVERSİTESİ
CERRAHPAŞA

2024-04-28

BIMU2061 – File Organization

Table of Contents

1. Introduction

The purpose of this project is to index text files under a specific folder by reading them.

The project code reads the passwords in each line of the text files (without reading repeated passwords), processes these passwords in a specific format, and saves them to a different folder, thus aiming to speed up the search function.

In line with the project objectives, cryptographic functions (MD5, SHA256, etc.) are used to process each password.

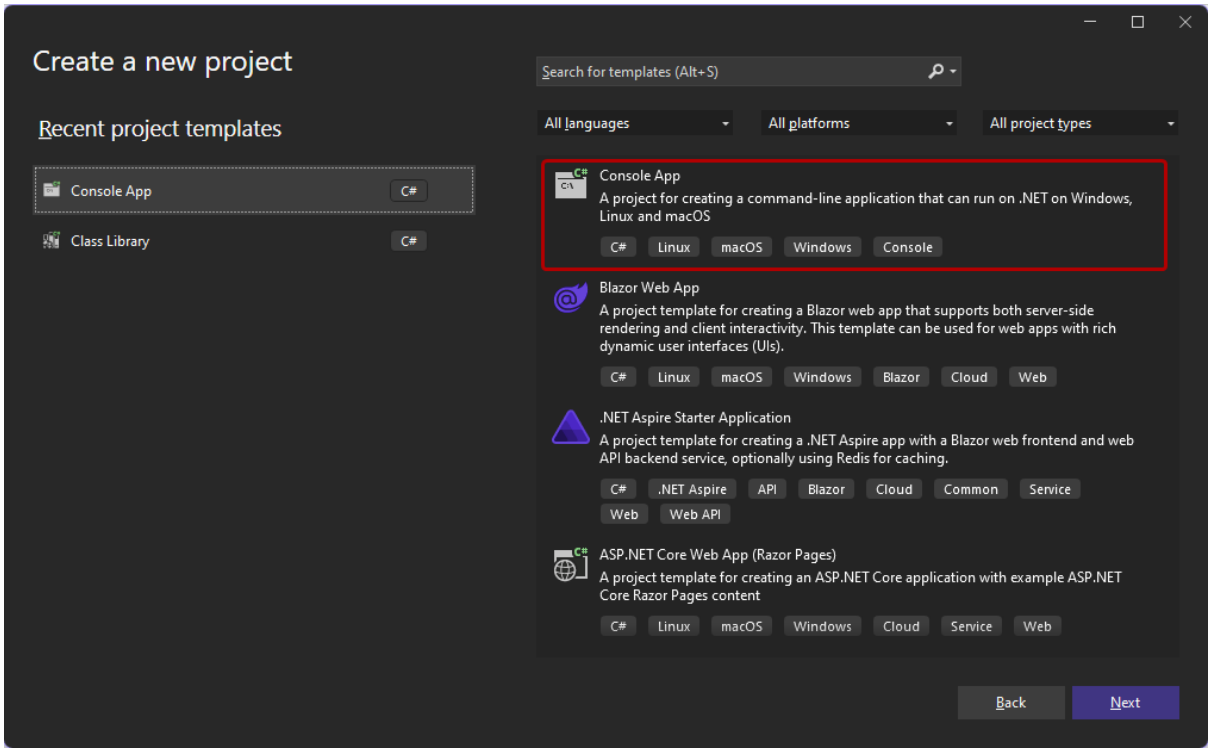
2. Tools and Technologies Used

The code was written in C# language, according to .NET version 8.0 and C# version 12.

Visual Studio 2022 Preview version 17.10.0 was used as the code editor.

The core functionality layer of the project, **SecureList.Core**, was created using the Class Library template; for running the project on the console screen, **SecureList.Console** was created using the Console App template.

No third-party libraries were used aside from .NET standard class libraries.



3. Project Design and Configuration

The folder structure and configuration of the project are detailed in the image below.

4. Indexing Process

Indexing Function

The indexing process is implemented within the **IndexService** class in the **SecureList.Core** project. This class utilizes a private property named *_passwordRepository*, which internally uses the *Dictionary<TKey,TValue>* structure. This data structure is highly efficient for search operations.

Let's elaborate on the operations within the **IndexService** class:

1. The `_passwordsRepository.Passwords` property stores passwords as key-value pairs in a *Dictionary* data structure, where the key represents the password and the value represents the folder where the password is located.
2. The `Index()` method of the **IndexService** class groups the current password and file name pairs by their initial characters. This operation is performed using the GroupBy() method from LINQ.
3. For each group:
   - An anonymous type consisting of password and file name pairs is created.
   - MD5, SHA1, and SHA256 hashes are calculated for the password. This function is performed by the `CryptographyHelper.CalculateHashes(string password)` method.
   - Each pair is concatenated into a string, separated by the "|" character, including the password, hashes, and file name.
4. The generated strings are divided into subgroups based on the `StaticDetails.MaximumLineNumberInIndexFiles` number (i.e., 10,000). This operation is performed using the Chunk() method from LINQ.
5. For each subgroup, a directory is created starting with the respective character, and a file name is assigned to this directory. Subsequently, the content of each subgroup (each chunk) is written to the respective subgroup file.

Functionality for Reading the Index

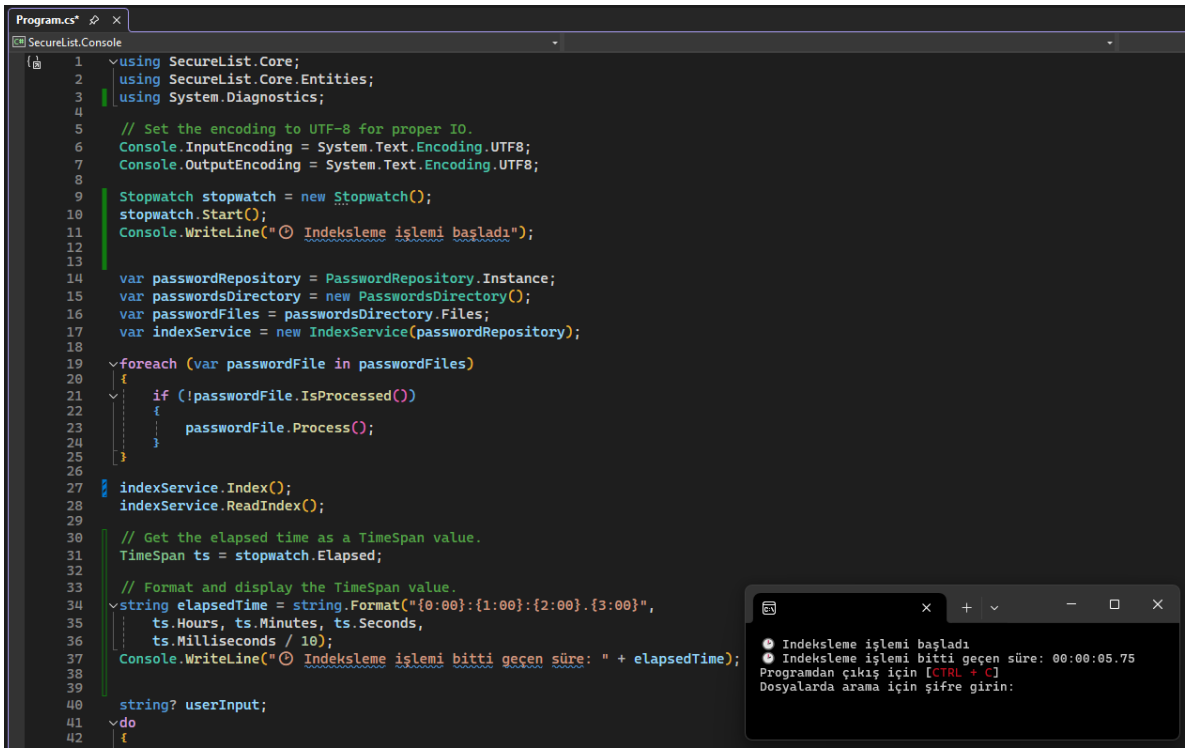
1. The `ReadIndex` method takes the directory where the index is located.
2. If the directory exists, it retrieves all file paths in that directory. Then, for each file path, it reads each line in the file.
3. Each line is split by the "|" character to separate the password and file name.
4. This password and file name are stored using the `TryAddPassword()` method of the `_passwordRepository` object.

These steps describe the process of indexing by taking password and file name pairs from a given Dictionary and then reading this index.

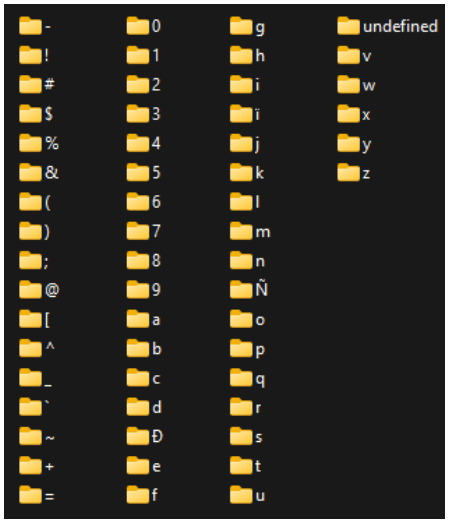
5. Search Function and Performance Testing

Searching with the Dictionary<TKey, TValue> data type has a time complexity very close to O(1) because it employs a hash table implementation. Therefore, performing searches in this data structure will yield very fast results.

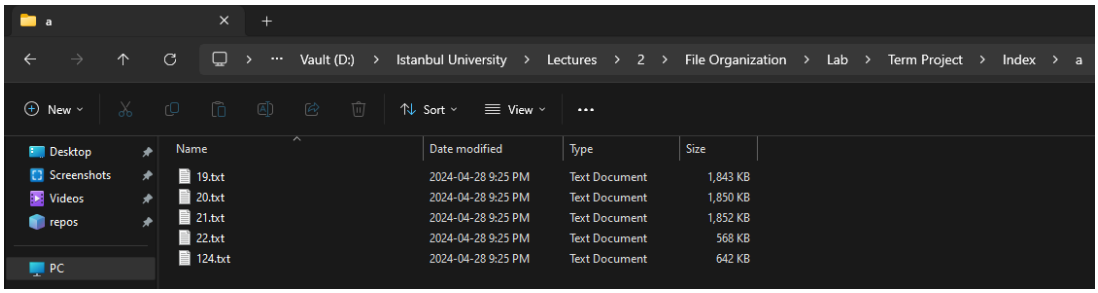
The indexing process takes 00:00:05.75 seconds



After indexing, the contents of the Index folder are as follows:

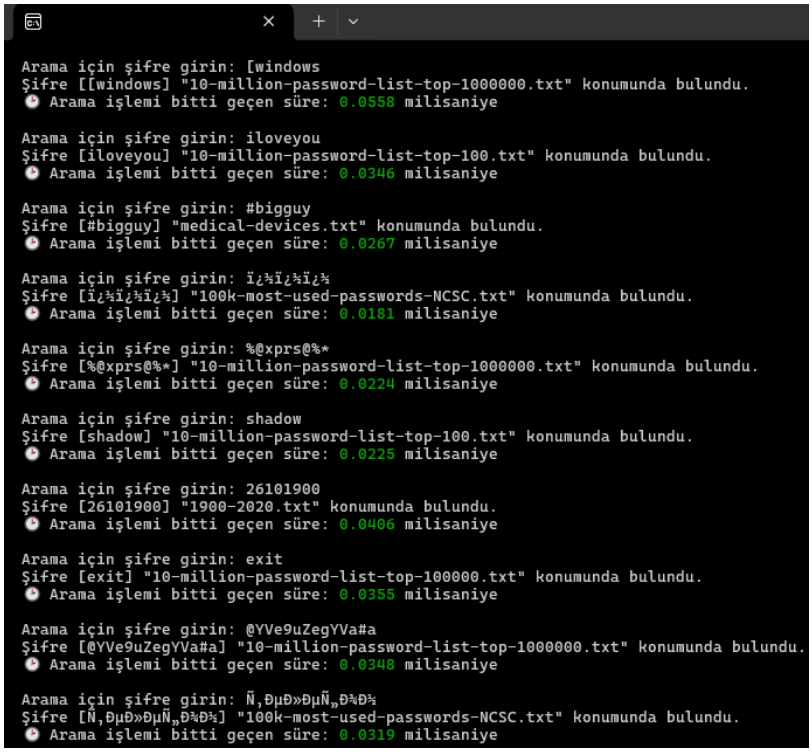


After indexing, the folder containing passwords starting with ***a*** or ***A*** within the Index directory.



After the indexing process, here are 10 randomly searched passwords and their retrieval times:

Searched Password	Retrieval Time (milliseconds)
[windows	0.0558
iloveyou	0.0346
#bigguy	0.0267
ï¿½ï¿½ï¿½	0.0181
%@xprs@%*	0.0224
shadow	0.0225
26101900	0.0406
exit	0.0355
@YVe9uZegYVa#a	0.0348
Ñ‚ÐµÐ»ÐµÑ„Ð¾Ð½	0.0319
	Average: 0.0323



6. Results and Evaluation

🧠 This project was quite educational because it contained example information about C# LINQ methods, anonymous types, and data structures.

💪🏻 It was helpful to see the speed of the *Dictionary* structure and to better understand the importance of indexing for large datasets. From this perspective, the *Dictionary* structure can certainly be used in search functions in future projects.

🎓 It was a project that demonstrated how a comprehensive project can be broken down into small parts and methods, making it easier to handle each function with object-oriented programming.

Looking at the project from a broad perspective, it touched on different topics, such as the function of determining whether files can be created with invalid characters in the Windows file system. Therefore, the project could be revisited for reference in the future.

7. References

The reference site and C# documentation used were learn.microsoft.com.

1. https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/types/anonymous-types
2. https://stackoverflow.com/questions/5750203/how-to-write-unicode-characters-to-the-console
3. https://learn.microsoft.com/en-us/dotnet/api/system.diagnostics.stopwatch.elapsed?view=net-8.0#examples