

SECURE CLOUD STORAGE




The language I used for this project is C#. My application is a simple version of how a secure cloud storage, such as Google Drive or DropBox, works. The application will secure all files uploaded to the cloud, such that only people who have gained the right to download or are part of the groups that have gained access will be able to decrypt uploaded files. Other members cannot download the file. If they try to access the file by force, they will only be able to get the encrypted version of the file.

A. Authentication	2
1. User Registration	2
2. Sign In:	4
B. Authorization	4
1. File Management	5
a. Upload File	5
b. Display Uploaded File	9
2. Encryption and Decryption	13
a. Encrypt Uploaded Files:	13
b. Decrypt File:	15
c. Save and Load AES Key for General Usage Purposes:	16
3. Group Management	17
a. Create A New Group:	17
b. Display groups:	19
c. Edit And Delete A Group:	21
d. Upload a File and Share To Groups:	24
4. Reset The Cloud	26
C. Security Features	27
D. Future Improvements	28
E. Appendix	28

A. Authentication

Secure Cloud Storage Home Register New User Sign in

Cloud Storage Application

A secure cloud storage application for Dropbox , Google Drive , Office365 .

© 2025 - Fay Nguyen

Landing Page of Secure Cloud Storage

1. User Registration

A new user needs to register their identification first to use the Secure Cloud Storage. They will be prompted to fill in some personal information: their first and last name, sign-up email, and the password for authentication purposes.

Secure Cloud Storage Home Register New User Sign in

Register A New User

First Name

Last Name

Email

Password

Register

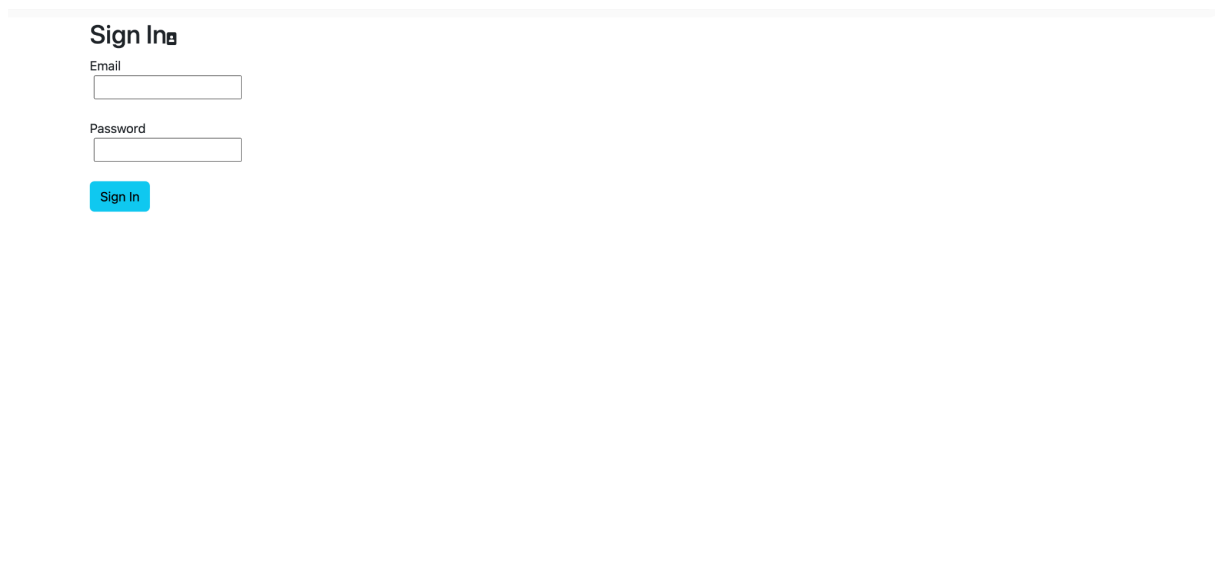
The inputted information will be passed to the **GenerateUserCertificate** service (**SecureCloudStorage/Application/**) to generate an X.509 certificate for the user. Firstly, a new 2048-bit RSA key is generated to sign the future certificate. Utilizing the package **X500DistinguishedName** of C#, the app generates a distinguished name (a unique identifier that represents the user in the cloud server used for authenticating users, managing access controls, and identifying specific entries in a directory) for the user. Then, the application creates an X.509 certificate request using the user information (email + full name), the RSA key generated previously, the SHA256 hash algorithm, and PKCS#1 padding. We must declare that this certificate is self-signed (not signed by any Certificate Authority) as well as a regular end-user certificate for encryption and decryption. The request will result in a self-signed certificate issued for the user that is valid for 1 month. Regarding the newly generated certificate, it extracts the user's the public key, which will be stored in a **.cer** file (a public key certificate), and the PKCS#12 archive (**.pfx**) that stores the associated private key, which will be protected by a password and then saved in the file for each user.

```
public UserCertificate GenerateUserCertificate(string email, string firstName,
string lastName){
    using var rsa = RSA.Create(2048);
    var subject = new X500DistinguishedName($"CN={firstName} {lastName},
E={email}");
    var request = new CertificateRequest(subject, rsa, HashAlgorithmName.SHA256,
RSASignaturePadding.Pkcs1);
    request.CertificateExtensions.Add(new X509BasicConstraintsExtension(false,
false, 0, false));
    var cert = request.CreateSelfSigned(DateTimeOffset.Now,
DateTimeOffset.Now.AddMonths(1));
    var publicBytes = cert.Export(X509ContentType.Cert);
    var privateBytes = cert.Export(X509ContentType.Pkcs12, "Stolaf");
    return new UserCertificate{
        FullName = firstName + lastName,
        Email = email,
        PublicKey = publicBytes,
        PrivateKey = privateBytes
    };
}
```

Finally, the table **User** of the database records the information of the newly created user: last name, first name, email, password, and the public key.

```
var user = new User
{
    Email = model.Email,
    FirstName = model.FirstName,
    LastName = model.LastName,
    PublicKey = userCert.PublicKey,
    Password = model.Password
};
```

2. Sign In:

A screenshot of a web form titled "Sign In". The form is contained within a light gray rectangular box. It features two input fields: the first is labeled "Email" and the second is labeled "Password". Below these fields is a blue button with the text "Sign In" in white. The form is set against a background of horizontal wavy lines in shades of blue and white.




To conduct further actions, users must sign in using their email and the password they have registered. The application will find the corresponding user stored in the **User** table and check whether the input password is correct. If the user can successfully sign in, the app will create an HTTP session to maintain the user's identity and to store user-specific data during future request/response interactions.

```
HttpContext.Session.SetString("User", user.FirstName);  
HttpContext.Session.SetString("Email", user.Email);  
HttpContext.Session.SetInt32("Id", user.Id);
```

B. Authorization

After the authentication step, the app will direct the user to a new home page.

Cloud Storage Application

A secure cloud storage application for Dropbox , Google Drive , Office365 .

1. File Management

Users can either upload a new file or display all the uploaded files so far.

a. Upload File

The current sign-in user can upload a file from their computer (one file for each upload). Then, they can type in the email addresses of the registered users to whom they want to give access or choose to share the file with at least one group. The file will be encrypted by the **EncryptionService**. The server will only store the encrypted version, not the original file, as well as the required data for accessible users to decrypt when they want to download the file. These will be stored in a secure server-side storage that the client side cannot access (**SecureCloudStorage/Infrastructure/Storage/**).

Browse File From Your Computer:

No file chosen

Enter emails of Recipients you want to gain access(comma/new line/semi-colon separated emails):

Or choose a group of users:

The function **Upload** handles the file submission when the user submits the upload form. It makes sure every user and user group (including themselves) that the uploader inputs will gain access to the encrypted file. The encryption is conducted by the function **EncryptAndSaveFile**. It stores the file access of the assigned groups and the current file in the **GroupFileAccess** table. If the upload succeeds, the application will direct to the successful upload page.



File Uploaded and Encrypted Successfully!

The file has been uploaded and encrypted securely.

```
[HttpPost]
public async Task<IActionResult> Upload(EncryptedFileViewModel model)
{
    var email = HttpContext.Session.GetString("Email");
    if (string.IsNullOrEmpty(email))
    {
        TempData["Error"] = "You must be logged in to upload files.";
        return RedirectToAction("Signin", "Signin");
    }

    var uploader = _context.Users.FirstOrDefault(u => u.Email == email);
    if (uploader == null)
```

```

    {
        TempData["Error"] = "Uploader account not found.";
        return RedirectToAction("Signin", "Signin");
    }

    var storageBase = Path.Combine(Directory.GetCurrentDirectory(),
    "../SecureCloudStorage.Infrastructure", "Storage");

    //a recipient email is inputted
    if (model.SelectedGroupIds == null &&
    (!string.IsNullOrEmpty(model.RecipientEmails) && model.SelectedGroupIds.Count
    == 0)) return View(model);

    var emails = new List<string>();
    if (!string.IsNullOrEmpty(model.RecipientEmails))
        emails = emails.Concat(model.RecipientEmails.Split(new[] {',', ';',
    '\n'}, StringSplitOptions.RemoveEmptyEntries).Select(emails
    =>emails.Trim()).ToList()).ToList();

    var groupMembers = new List<User>();
    //return error if the emails are not in the database
    //add the user of the group into the emails
    foreach (var group in model.SelectedGroupIds){
        groupMembers = groupMembers.Concat(_context.GroupMembers
        .Where(u => u.GroupId == group).Select(u => u.User).ToList()).ToList();
    }

    var recipients = _context.Users
        .Where(u => emails.Contains(u.Email))
        .ToList();

    recipients = recipients.Concat(groupMembers).ToList();
    var foundEmails = recipients.Select(r => r.Email).ToHashSet();
    var missingEmails = emails.Where(e => !foundEmails.Contains(e)).ToList();
    recipients = recipients.Concat(new List<User>{uploader}).
        GroupBy(u => u.Email).
        Select(g => g.First()).
        ToList();

    if (missingEmails.Any())
    {
        ViewBag.MissingRecipients = "Invalid Emails: " + string.Join(", ",
    missingEmails);
    }

    var file = model.File;
    //Without await, the method returns a Task<EncryptedFile>, not the actual
    EncryptedFile, which leads to runtime issues
    var encryptedFile = await EncryptAndSaveFile(file, recipients, uploader.Id);

    foreach (var groupId in model.SelectedGroupIds ?? new List<int>())

```

```

    {
        _context.GroupFileAccesses.Add(new GroupFileAccess
        {
            GroupId = groupId,
            FileId = encryptedFile.Id,
        });
    }
    await _context.SaveChangesAsync();

    //show the upload successfully page
    return RedirectToAction("UploadSuccessfully");
}

```

The helper function **EncryptAndSaveFile** is responsible for encrypting the uploaded file, saving the encrypted file and its associated metadata (including the AES initialization vector and file name) to server-side storage, recording file details — file name, uploader's ID, upload timestamp, metadata path, and encrypted file path — into the **EncryptedFile** table, as well as assigning and recording access for each authorized user by saving entries into the **UserFileAccess** table, linking the encrypted file with the users allowed to decrypt it.

```

private async Task<EncryptedFile> EncryptAndSaveFile(IFormFile file, List<User>
recipients, int uploaderId)
{
    var storageBase = Path.Combine(Directory.GetCurrentDirectory(),
"..\\SecureCloudStorage.Infrastructure", "Storage");

    var filePath = Path.Combine(storageBase, "uploads", $"{file.FileName}.enc");
    var metadataPath = Path.Combine(storageBase, "metadata",
$"{file.FileName}.meta.json");

    //read file into bytes
    using var ms = new MemoryStream();
    await file.CopyToAsync(ms);
    var fileBytes = ms.ToArray();

    var (encryptedFile, metadata) =
_encryptionService.EncryptFile(file.FileName, fileBytes, recipients);

    Directory.CreateDirectory(Path.GetDirectoryName(filePath)!);
    System.IO.File.WriteAllBytes(filePath, encryptedFile);

    Directory.CreateDirectory(Path.GetDirectoryName(metadataPath)!);
    System.IO.File.WriteAllText(metadataPath,
JsonSerializer.Serialize(metadata));

    var encryptedFileEntry = new EncryptedFile

```



```
{
    FileName = file.FileName,
    EncryptedPath = filePath,
    MetadataPath = metadataPath,
    UploadedAt = DateTime.UtcNow,
    UploaderId = uploaderId
};

_context.EncryptedFiles.Add(encryptedFileEntry);
await _context.SaveChangesAsync();

foreach (var recipient in recipients)
{
    _context.UserFileAccesses.Add(new UserFileAccess
    {
        UserId = recipient.Id,
        FileId = encryptedFileEntry.Id,
        EncryptedAesKey = metadata.EncryptedKeys[recipient.Email]
    });
}

await _context.SaveChangesAsync();

return encryptedFileEntry;}
```

b. Display Uploaded File

The application provides a view of all uploaded files. If the currently signed-in user has been granted access to a specific file, they are allowed to download and decrypt it. Otherwise, the application will display a warning indicating that the file cannot be decrypted due to lack of access. Even if an unauthorized user attempts to bypass the system—such as by hacking or directly accessing server-side storage—they will only be able to retrieve the encrypted version of the file, which remains unreadable without proper decryption.

Secure Cloud Storage

Home

Upload File

File Uploaded Gallery

Add A New Group

Display Groups

Private Tab!!!

Welcome, Khanh

Log Out

File Name	Uploader	Uploaded At	Action
download.png	Khanh Tran	12/04/2025 00:04	<div>Download</div>
Grizzly1.jpg	Khanh Tran	12/04/2025 13:49	<div>Download</div>

Display File View of A User With Access

Secure Cloud Storage

Home

Upload File

File Uploaded Gallery

Add A New Group

Display Groups

Private Tab!!!

Welcome, Grizzly

Log Out

File Name	Uploader	Uploaded At	Action
download.png	Khanh Tran	12/04/2025 00:04	No permission to decrypt and download
Grizzly1.jpg	Khanh Tran	12/04/2025 13:49	No permission to decrypt and download

Display File View of A User Without Access

```
public IActionResult DisplayFiles() {
    var userEmail = HttpContext.Session.GetString("Email");
    var files = _context.EncryptedFiles.Include(f => f.Uploader)
        .Include(f => f.AccessList)
        .Select(file => new FileDisplayViewModel
        {
            FileId = file.Id,
            FileName = file.FileName,
            UploadedAt = file.UploadedAt,
```

```
UploaderName =  
file.Uploader.FirstName + " " + file.Uploader.LastName,  
Downloadable = (userEmail != null)  
&& file.AccessList.Any(a => a.User.Email == userEmail)  
}).ToList();  
  
return View(files);  
}
```

For the accessible users, they can download the file. The function handles the secure file download process. It begins by verifying that the user is signed in and has permission to access the requested file. If authorized, the application locates the encrypted file and the user's private key from the server. It then reads the corresponding metadata and decrypts the file using the AES key — retrieved through RSA decryption with the user's private key. If successful, the decrypted file is returned for download; otherwise, appropriate error messages are shown. This ensures that only authorized users can access and decrypt files securely.

```
public IActionResult Download(int id)  
{  
    var userEmail = HttpContext.Session.GetString("Email");  
    Console.WriteLine($"File ID: {id}");  
  
    if (userEmail == null)  
    {  
        return RedirectToAction("Signin", "Signin");  
    }  
  
    var user = _context.Users.FirstOrDefault(u => u.Email == userEmail);  
    if (user == null)  
    {  
        ViewBag.error = "Access denied";  
        return View();  
    }  
  
    var access = _context.UserFileAccesses.FirstOrDefault(x => x.UserId ==  
user.Id && x.FileId == id);  
    if (access == null)  
    {  
        ViewBag.error = "Access denied";  
        return Unauthorized("No permission to decrypt and download");  
    }  
  
    var file = _context.EncryptedFiles.FirstOrDefault(f => f.Id == id);  
    if (file == null)  
    {  
        ViewBag.error = "File Not Found";  
        return NotFound("File not found.");  
    }  
}
```

```
}

if (!System.IO.File.Exists(file.EncryptedPath))
{
    return NotFound("Encrypted file does not exist on server.");
}

var encryptedBytes = System.IO.File.ReadAllBytes(file.EncryptedPath);

var privateKeyPath = Path.Combine(
    Directory.GetCurrentDirectory(),
    "../SecureCloudStorage.Infrastructure/Storage/certs-private",
    $"{user.Email}.pfx"
);

if (!System.IO.File.Exists(privateKeyPath))
{
    ViewBag.error = "Private key not found for this user";
    return NotFound("Private Key Not Found");
}

var privateKey = System.IO.File.ReadAllBytes(privateKeyPath);

var jsonString = System.IO.File.ReadAllText(file.MetadataPath);

FileMetadata metadata;
try
{
    metadata = JsonSerializer.Deserialize<FileMetadata>(jsonString);
}
catch (Exception ex)
{
    return BadRequest($"Failed to parse metadata: {ex.Message}");
}

if (metadata == null || metadata.InitializationVector == null)
{
    return BadRequest("Invalid or missing metadata");
}

byte[] decryptedBytes;
try
{
    decryptedBytes = _encryptionService.DecryptFile(
        encryptedBytes,
```

```
        metadata.InitializationVector,  
        access.EncryptedAesKey,  
        privateKey  
    );  
}  
catch (Exception ex)  
{  
    return BadRequest($"Decryption failed: {ex.Message}");  
}  
  
return File(decryptedBytes, "application/octet-stream", file.FileName);  
}
```

2. Encryption and Decryption

The service **EncryptionService (SecureCloudStorage.Application/)** controls the encryption and decryption activities of the cloud storage. This service has 3 main functions.

a. Encrypt Uploaded Files:

```
public (byte[] EncryptedFile, FileMetadata Metadata) EncryptFile(string fileName,  
byte[] fileData, List<User> recipients);
```

The function **EncryptFile** of the service **EncryptionService (SecureCloudStorage.Application/)** controls the encryption of the files. It requires the name, the uploaded file's data (converted to bytes), and the list of users that receive access to the file after encryption.

```
using var aes = Aes.Create();  
aes.GenerateIV();  
aes.GenerateKey();  
//Encrypt the file  
//using to make sure that ms is properly disclosed (cleaned up after finish)  
using var ms = new MemoryStream();  
//parameter: ms - the output stream, aes.CreateEncryptor() - create an  
encryptor based on the initialization vector and AES key; CryptoStreamMode.Write -  
mode that plaintext is written to the crypto stream and it writes bytes to the ms  
using var cs = new CryptoStream(ms, aes.CreateEncryptor(),  
CryptoStreamMode.Write);  
cs.Write(fileData, 0, fileData.Length);  
cs.FlushFinalBlock();
```

It first creates a new AES encryption instance and randomly generates an initialization vector and a secret AES key. The server creates an in-memory (temporary) stream **ms** to store the data after encryption and a special Cryptostream **cs** that applies the encryption/decryption algorithm on data while it is read or written. After it finishes transferring the encrypted data to the temporary stream, it converts the temporary stream into an array to store the encrypted file bytes.

```
//Encrypt RSA key for each user
var encryptedKeys = new Dictionary<string, byte[]>();
foreach (var user in recipients){
    var cert = new X509Certificate2(user.PublicKey);
    using var rsa = cert.GetRSAPublicKey();
    var encryptedKey = rsa.Encrypt(aes.Key,
RSAEncryptionPadding.OaepSHA256);
    encryptedKeys[user.Email] = encryptedKey;
}

var storageBase = Path.Combine(Directory.GetCurrentDirectory(),
"..\\SecureCloudStorage.Infrastructure", "Storage");
var aesKeyPath = Path.Combine(storageBase, "aeskeys", $"{fileName}.enc");

SaveEncryptedAESKey(aes.Key, aesKeyPath, _aes_service.GetMasterKey());
```

Regarding the AES key it generated for encryption, we need to store this securely for decryption purposes or when we provide file access to other members. For each current user, the service loads their RSA public key from the X509 Certificate they gain during registration, uses RSA to encrypt the AES key, and stores their encrypted AES keys in the **UserFileAccess** table. In the future, if the users' file permission is revoked, the application will directly delete the corresponding entry in the database, so there will not be an AES key for them to decrypt the file. This ensures that only users who have access to the uploaded files can decrypt and download them from the cloud storage. Besides, for general purposes (backup or adding new user's access later), I decided to store the AES key separately and securely. I use another service, **SaveEncryptedAESKey**, to finish this task and store it on the server-side storage (users cannot access from the client side).

```
public void SaveEncryptedAESKey(byte[] aesKey, string aesKeyFilePath, byte[]
masterKey)
{
    using var aes = Aes.Create();
    aes.Key = masterKey;
    //Improve: randomly generate a new IV
    aes.IV = new byte[16];

    using var encryptor = aes.CreateEncryptor();
    var encryptedKey = encryptor.TransformFinalBlock(aesKey, 0,
aesKey.Length);
```

```
Directory.CreateDirectory(Path.GetDirectoryName(aesKeyFilePath)!);
using var fs = new FileStream(aesKeyFilePath, FileMode.Create,
FileAccess.Write);
fs.Write(aes.IV, 0, aes.IV.Length);
fs.Write(encryptedKey, 0, encryptedKey.Length);
}
```

This service then treats the AES key as a file, using a predefined master key stored in the system as a new AES key and an initialization vector with 16 zero bits to encrypt and store it in the server side storage.

```
//ms.ToArray() is an array containing the encrypted file bytes
return (ms.ToArray(), new FileMetadata{
    FileName = fileName,
    InitializationVector = aes.IV,
    EncryptedKeys = encryptedKeys,
    AesKeyPath = aesKeyPath
});
```

After all the tasks, this service will return the bytes of the encrypted files and a file metadata containing the file name, the initialization vector used in the encryption phase, the users' encrypted keys, and the path to the file storing the encrypted AES key used for general purposes.

b. Decrypt File:

The function **Decrypt** of the service **EncryptionService (SecureCloudStorage.Application/)** controls the decryption of the files that users want to download. It requires the bytes of the encrypted data, the initialization vector used during encryption, and the encrypted AES key based on the user's RSA public key and the user's private key.

```
public byte[] DecryptFile (byte[] encryptedData, byte[] initializationVector, byte[]
encryptedAesKey, byte[] privateKey;
```

Regarding the encrypted AES key, the function needs to decrypt it first with the user's private key using the **DecryptAESKey**.

```
public byte[] DecryptAESKey(byte[] encryptedKey, byte[] privateKey)
{
    var cert = new X509Certificate2(privateKey, _secure_password,
X509KeyStorageFlags.Exportable);

    using var rsa = cert.GetRSAPrivateKey();
    return rsa.Decrypt(encryptedKey, RSAEncryptionPadding.OaepSHA256);
}
```

With the newly decrypted AES key and the initialization vector, the function creates an AES instance to utilize its decryptor to decrypt the encrypted file's byte following the same method of using an in-memory stream and a crypto stream to store and apply the decryption algorithm to the decrypted data. Finally, it returns the temporary decrypted data stored in the array.

```
using var aes = Aes.Create();
aes.Key = DecryptAESKey(encryptedAesKey, privateKey);
aes.IV = initializationVector;
using var ms = new MemoryStream(encryptedData);
using var cs = new CryptoStream(ms, aes.CreateDecryptor(),
CryptoStreamMode.Read);
using var output = new MemoryStream();
cs.CopyTo(output);
return output.ToArray();
```

c. Save and Load AES Key for General Usage Purposes:

When we need to add a new member to a group that already has encrypted file access, we also need to give the new member their own encrypted AES key. This is one of the main reasons I decided to store each encrypted file's AES key in a secure server-side storage (**SecureCloudStorage.Infrastructure/Storage/aeskeys/**). I added one more security layer by encrypting it with a predefined master key and a zero-value 16-byte initialization vector.

The function **SaveEncryptedAESKey** encrypts the AES key and then stores it in a file in the following format: 16-byte initialization vector first + data of the encrypted AES key.

```
public void SaveEncryptedAESKey(byte[] aesKey, string aesKeyFilePath, byte[]
masterKey)
{
    using var aes = Aes.Create();
    aes.Key = masterKey;
    aes.IV = new byte[16];

    using var encryptor = aes.CreateEncryptor();
    var encryptedKey = encryptor.TransformFinalBlock(aesKey, 0, aesKey.Length);

    Directory.CreateDirectory(Path.GetDirectoryName(aesKeyFilePath)!);
    using var fs = new FileStream(aesKeyFilePath, FileMode.Create,
FileAccess.Write);
    fs.Write(aes.IV, 0, aes.IV.Length);
    fs.Write(encryptedKey, 0, encryptedKey.Length);
}
```

The function **LoadDecryptedAESKey** reads the AES key file, decrypts it using an AES instance, and returns the original AES key.


```
public byte[] LoadDecryptedAESKey(string aesKeyFilePath, byte[] masterKey)
{
    var allBytes = System.IO.File.ReadAllBytes(aesKeyFilePath);
    var iv = allBytes.Take(16).ToArray();
    var encryptedKey = allBytes.Skip(16).ToArray();


    using var aes = Aes.Create();
    aes.Key = masterKey;
    aes.IV = iv;

    using var decryptor = aes.CreateDecryptor();
    return decryptor.TransformFinalBlock(encryptedKey, 0, encryptedKey.Length);
}
```

3. Group Management

Besides granting access to users individually, the uploader can also share access to a group so every member in the group will have access.

a. Create A New Group:

Add A New Group 

Group Name

Enter emails of the member Group (comma/new line/semi-colon separated emails):

Enter emails of admins (comma/new line/semi-colon separated emails):

The current user is prompted to fill in some information to create a new group (group name, emails of the members, emails of admin members). The creator will be automatically included in the group as well as become an admin. The new group will be recorded into the **GroupMember** table, and the corresponding members of each group will be recorded into the **UserMember** table for future usage.

```
//POST /Files/Upload
//handle the file submission when the user submits the upload form
```

```

[HttpPost]
public async Task<IActionResult> AddGroup(AddGroupViewModel model)
{
    var curr_user = _context.Users
        .Where(u => u.Email == HttpContext.Session.GetString("Email"))
        .ToList();

    var emails = new List<string>();
    var adminemails = new List<string>();
    if(!String.IsNullOrEmpty(model.MemberEmails)) emails =
model.MemberEmails.Split(new[] {',', ';', '\n'},
StringSplitOptions.RemoveEmptyEntries).Select(emails =>emails.Trim()).ToList();
    if(!String.IsNullOrEmpty(model.AdminEmails)) adminemails =
model.AdminEmails.Split(new[] {',', ';', '\n'},
StringSplitOptions.RemoveEmptyEntries).Select(emails =>emails.Trim()).ToList();

    //return error if the emails are not in the database
    var members = _context.Users
        .Where(u => emails.Contains(u.Email))
        .ToList();
    var admins = _context.Users
        .Where(u => adminemails.Contains(u.Email))
        .ToList();

    members = members.Concat(curr_user).GroupBy(u =>
u.Id).Select(g=>g.First()).ToList();
    admins = admins.Concat(curr_user).GroupBy(u =>
u.Id).Select(g=>g.First()).ToList();

    foreach(var user in admins){
        if(!members.Contains(user)) admins.Remove(user);
    }

    var foundEmails = members.Select(r => r.Email).ToHashSet();
    var missingEmails = emails.Where(e =>
!foundEmails.Contains(e)).ToList();

    members = members.Concat(curr_user).
        GroupBy(u => u.Email).
        Select(g => g.First()).
        ToList();

    if (missingEmails.Any())
    {
        ViewBag.MissingRecipients = "Emails not exist: " + string.Join(",
", missingEmails);
    }

    await _context.SaveChangesAsync();

    //Add a new group
    _context.Groups.Add(new Group{

```

```

        Name = model.GroupName
    });
    await _context.SaveChangesAsync();
    var currGroup = _context.Groups.Where(u => u.Name ==
model.GroupName).ToList()[0];
    foreach (var member in members)
    {
        _context.GroupMembers.Add(new GroupMember
        {
            UserId = member.Id,
            GroupId = currGroup.Id,
            Admin = admins.Contains(member)
        });
    }
    await _context.SaveChangesAsync();

    //show the upload successfully page

    return RedirectToAction("DisplayGroup");
}

```

b. Display groups:

Group Name	Members	
Uzumaki Clan	Khanh Tran (knguyent@tcd.ie) Kha Trinh (minhkha01@gmail.com) Grizzly WeBareBears (grizzly1@gmail.com)	Edit Delete
We Bare Bears	Grizzly WeBareBears (grizzly1@gmail.com) Panda WeBareBears (Panda2@gmail.com) IceBear WeBareBears (icebear@gmail.com)	Edit Delete

If the group is created successfully, the application will direct to the view that displays all the groups created. Each group entry lists all the names of the members as well as their emails. If the current user is an admin of the group, they can have options to modify the group (either edit or delete the group permanently). Now, the uploaded files can be shared with groups of members:

```

public IActionResult DisplayGroup() {
    var curr_user = _context.Users
        .Where(u => u.Email == HttpContext.Session.GetString("Email"))
        .ToList();

    var memberList = _context.Groups.Include(g => g.GroupMembers).Select(group
=> new{
        GroupName = group.Name,
        Members = group.GroupMembers.Select(m => new{

```

```
                First = m.User.FirstName, Last = m.User.LastName, Email
= m.User.Email
            })
        });
        var admin = _context.GroupMembers.Where(member => member.UserId ==
curr_user[0].Id).Select(group => group.GroupId).ToList();
        var memberEmails = new Dictionary<string, string>();
        foreach (var group in memberList){
            var tempString = "";
            foreach(var member in group.Members){
                tempString += member.First + " " + member.Last + " (" + member.Email
+ ") \n";
            }
            memberEmails[group.GroupName] = tempString;
        }
        var groups = _context.Groups.Select(group => new DisplayGroupViewModel
        {
            GroupName = group.Name,
            MemberEmails = memberEmails[group.Name],
            Admin = admin.Contains(group.Id),
            Id = group.Id
        }).ToList();

        return View(groups);
    }
}
```

c. Edit And Delete A Group:

Edit The Group

Group Name

Uzumaki Clan

[Edit](#)

Enter emails of the member Group (comma/new line/semi-colon separated emails):

knguyent@tcd.ie minhkha01@gmail.com grizzly1@gmail.com

[Edit](#)

Enter emails of admins (comma/new line/semi-colon separated emails):

knguyent@tcd.ie grizzly1@gmail.com

[Edit](#)

Edit Group

Admin members are eligible to edit the group. They can rename the group, add or remove group members, and modify the list of admins. All of the changes will be updated to the database accordingly. If they remove a current member, all of the files that the removed member gains access to from the group will be inaccessible, while the new members will gain access to the file with their encrypted AES key using the AES key we store securely before. The function below will add access to the new members and generate their own encrypted AES keys.

```
private async Task GiveFileAccessToNewGroupMembers(List<User> newUsers, int
groupId)
{
    var storageBase = Path.Combine(Directory.GetCurrentDirectory(),
"..\\SecureCloudStorage.Infrastructure", "Storage");

    // Get all files that the group has access to
    var fileIds = _context.GroupFileAccesses
        .Where(gfa => gfa.GroupId == groupId)
        .Select(gfa => gfa.FileId)
        .ToList();

    foreach (var fileId in fileIds)
    {
        var file = _context.EncryptedFiles.FirstOrDefault(f => f.Id ==
fileId);

        if (file == null) continue;
```

```

        // Read metadata file
        var metadataPath = file.MetadataPath;
        if (!System.IO.File.Exists(metadataPath)) continue;

        var jsonString = await
System.IO.File.ReadAllTextAsync(metadataPath);
        var metadata =
JsonSerializer.Deserialize<FileMetadata>(jsonString);
        if (metadata == null || metadata.EncryptedKeys == null) continue;

        foreach (var user in newUsers)
        {
            if (_context.UserFileAccesses.Any(u => u.FileId == file.Id &&
u.UserId == user.Id))
                continue;

            // Encrypt AES key with the new user's public key
            var cert = new X509Certificate2(user.PublicKey);
            using var rsa = cert.GetRSAPublicKey();
            var encryptedKey =
rsa.Encrypt(_encryption_service.LoadDecryptedAESKey(metadata.AesKeyPath,
_aes_key_service.GetMasterKey()), RSAEncryptionPadding.OaepSHA256);

            _context.UserFileAccesses.Add(new UserFileAccess
            {
                FileId = file.Id,
                UserId = user.Id,
                EncryptedAesKey = encryptedKey
            });

            // Also update metadata (optional but nice to keep them in
sync)
            metadata.EncryptedKeys[user.Email] = encryptedKey;
        }

        // Save updated metadata
        await System.IO.File.WriteAllTextAsync(metadataPath,
JsonSerializer.Serialize(metadata));
    }

    await _context.SaveChangesAsync();
}

```

If a group is deleted, all of its access to encrypted files will be deleted, as well as its members' access to that file.



Delete Group!

The Group has been Delete.

Add A New Group

Show Current Groups

```
public async Task<IActionResult> DeleteGroup(int id)
{
    var filesIdAccessed = await _context.GroupFileAccesses
        .Where(u => u.GroupId == id)
        .Select(u => u.FileId)
        .ToListAsync();

    var usersIdAccessed = await _context.GroupMembers
        .Where(u => u.GroupId == id)
        .Select(u => u.UserId)
        .ToListAsync();

    // Remove file access entries for this group
    var groupFileAccesses = await _context.GroupFileAccesses
        .Where(g => g.GroupId == id)
        .ToListAsync();
    _context.GroupFileAccesses.RemoveRange(groupFileAccesses);

    // Remove users from the group
```

```
var userGroup = await _context.GroupMembers
    .Where(g => g.GroupId == id)
    .ToListAsync();
_context.GroupMembers.RemoveRange(userGroup);

// Remove user access to files that were accessed via this group
var fileUserAccess = await _context.UserFileAccesses
    .Where(u => usersIdAccessed.Contains(u.UserId) &&
filesIdAccessed.Contains(u.FileId))
    .ToListAsync();
_context.UserFileAccesses.RemoveRange(fileUserAccess);

// Remove the group itself
var group = await _context.Groups.FirstOrDefaultAsync(u => u.Id ==
id);

if (group != null)
{
    _context.Groups.Remove(group);
}

await _context.SaveChangesAsync();

TempData["Message"] = "✔ Group and related access have been
deleted.";

return RedirectToAction("DisplayGroup");
}
```

d. Upload a File and Share To Groups:

Browse File From Your Computer:

Choose File IceBear1.jpg

Enter emails of Recipients you want to gain access(comma/new line/semi-colon separated emails):

knguyent@tcd.ie

Or choose a group of users:

☒ We Bare Bears

Upload & Encrypt

When a group is given access to a file, every member will be able to decrypt it and download it.

Example: when Grizzly shares the file IceBear1.jpg to the group We Bare Bears, since Ice Bear is in that group, they now have permission to decrypt and download the file.

Secure Cloud Storage Home Upload File File Uploaded Gallery Add A New Group Display Groups Private Tab!!!			
Welcome, IceBear Log Out			
File Name	Uploader	Uploaded At	Action
download.png	Khanh Tran	12/04/2025 00:04	No permission to decrypt and download
Grizzly1.jpg	Khanh Tran	12/04/2025 13:49	No permission to decrypt and download
IceBear1.jpg	Grizzly WeBareBears	12/04/2025 15:29	Download

But if they delete the group We Bare Bears (Ice Bear is an admin), they are no longer able to decrypt the file and download it. It also applies to Panda and Grizzly, who are in the group.

Secure Cloud Storage Home Upload File File Uploaded Gallery Add A New Group Display Groups Private Tab!!!			
Welcome, IceBear Log Out			
Group Name		Members	

Secure Cloud Storage Home Upload File File Uploaded Gallery Add A New Group Display Groups Private Tab!!!			
Welcome, IceBear Log Out			
File Name	Uploader	Uploaded At	Action
download.png	Khanh Tran	12/04/2025 00:04	No permission to decrypt and download
Grizzly1.jpg	Khanh Tran	12/04/2025 13:49	No permission to decrypt and download
IceBear1.jpg	Grizzly WeBareBears	12/04/2025 15:29	No permission to decrypt and download

However, it does not apply to Khanh Tran, another user to whom Grizzly had granted access individually. This user still has permission to decrypt and download the file.

Secure Cloud Storage

Home

Upload File

File Uploaded Gallery

Add A New Group

Display Groups

Private Tab!!!

Welcome, Khanh

Log Out

File Name	Uploader	Uploaded At	Action
download.png	Khanh Tran	12/04/2025 00:04	<button>Download</button>
Grizzly1.jpg	Khanh Tran	12/04/2025 13:49	<button>Download</button>
IceBear1.jpg	Grizzly WeBareBears	12/04/2025 15:29	<button>Download</button>

4. Reset The Cloud

This is an add-on feature. Every time we want to reset the cloud storage (remove all the files uploaded, users, and groups), we can use the self-destruct button to wipe out all the data and start over again. This will clear all rows in the tables of the database, reset the auto-incremented IDs to 1, and delete all users' certificates, encrypted data, and metadata.

Self Destruct

Use this page to destroy everything -- What Dr. Doofenshmirtz always does with his invention!!! No one can stop you!

Self-Destruct



```
public async Task<IActionResult> WipeEverything()
{
    // 1. Delete files from disk
    var storageBase = Path.Combine(Directory.GetCurrentDirectory(),
    "../SecureCloudStorage.Infrastructure", "Storage");

    var dirs = new[] { "uploads", "metadata", "aeskeys", "certs-private",
    "certs"};
```

```

        foreach (var dir in dirs)
        {
            var fullPath = Path.Combine(storageBase, dir);
            if (Directory.Exists(fullPath))
            {
                Directory.Delete(fullPath, true);
            }
        }

        if(_context.UserFileAccesses.Count() > 0)
        _context.UserFileAccesses.RemoveRange(_context.UserFileAccesses);
        if(_context.GroupFileAccesses.Count() > 0)
        _context.GroupFileAccesses.RemoveRange(_context.GroupFileAccesses);
        if(_context.GroupMembers.Count() > 0)
        _context.GroupMembers.RemoveRange(_context.GroupMembers);
        if(_context.EncryptedFiles.Count() > 0)
        _context.EncryptedFiles.RemoveRange(_context.EncryptedFiles);
        if(_context.Groups.Count() > 0)
        _context.Groups.RemoveRange(_context.Groups);
        if(_context.Users.Count() > 0) _context.Users.RemoveRange(_context.Users);
        await _context.SaveChangesAsync();
        //clear everything before reset
        await _context.Database.ExecuteSqlRawAsync("ALTER TABLE `User`
AUTO_INCREMENT = 1;");
        await _context.Database.ExecuteSqlRawAsync("ALTER TABLE `GroupMember`
AUTO_INCREMENT = 1;");
        await _context.Database.ExecuteSqlRawAsync("ALTER TABLE `EncryptedFile`
AUTO_INCREMENT = 1;");
        HttpContext.Session.Remove("User");
        HttpContext.Session.Remove("Email");
        HttpContext.Session.Remove("Id");
        TempData["Message"] = "☢ Everything has been wiped!";

        return RedirectToAction("Index");
    }

```

C. Security Features

- **X509Certificate and X509Certificate2**
- **Encryption for Files:** Files are encrypted using the AES (Advanced Encryption Standard) package of C# before being stored. This ensures that the files are encrypted and decrypted using the same AES key (symmetric key) and AES initialization vector.
- **Encryption for encrypted files' AES Keys per user:** For each encrypted file, its AES key is encrypted per user who gains access using C# RSA (Rivest–Shamir–Adleman) package to encrypt the AES key with the user's public key and decrypt the AES key with their private

key gained through the X.509 certificate. This encrypted key is then recorded in the database and will be deleted if the access is revoked.

- **Metadata Management:** Each uploaded file has a metadata file storing the initialization vector of the encrypted file.
- **Master Key Encryption:** AES keys are additionally encrypted and stored using a master key for backup and other purposes.

D. Future Improvements

1. Host the cloud storage application online
2. Improve the UI (uploaded file preview)
3. Integrate the drag-and-drop upload
4. Implement an online database (currently using a local MySQL database)

E. Appendix

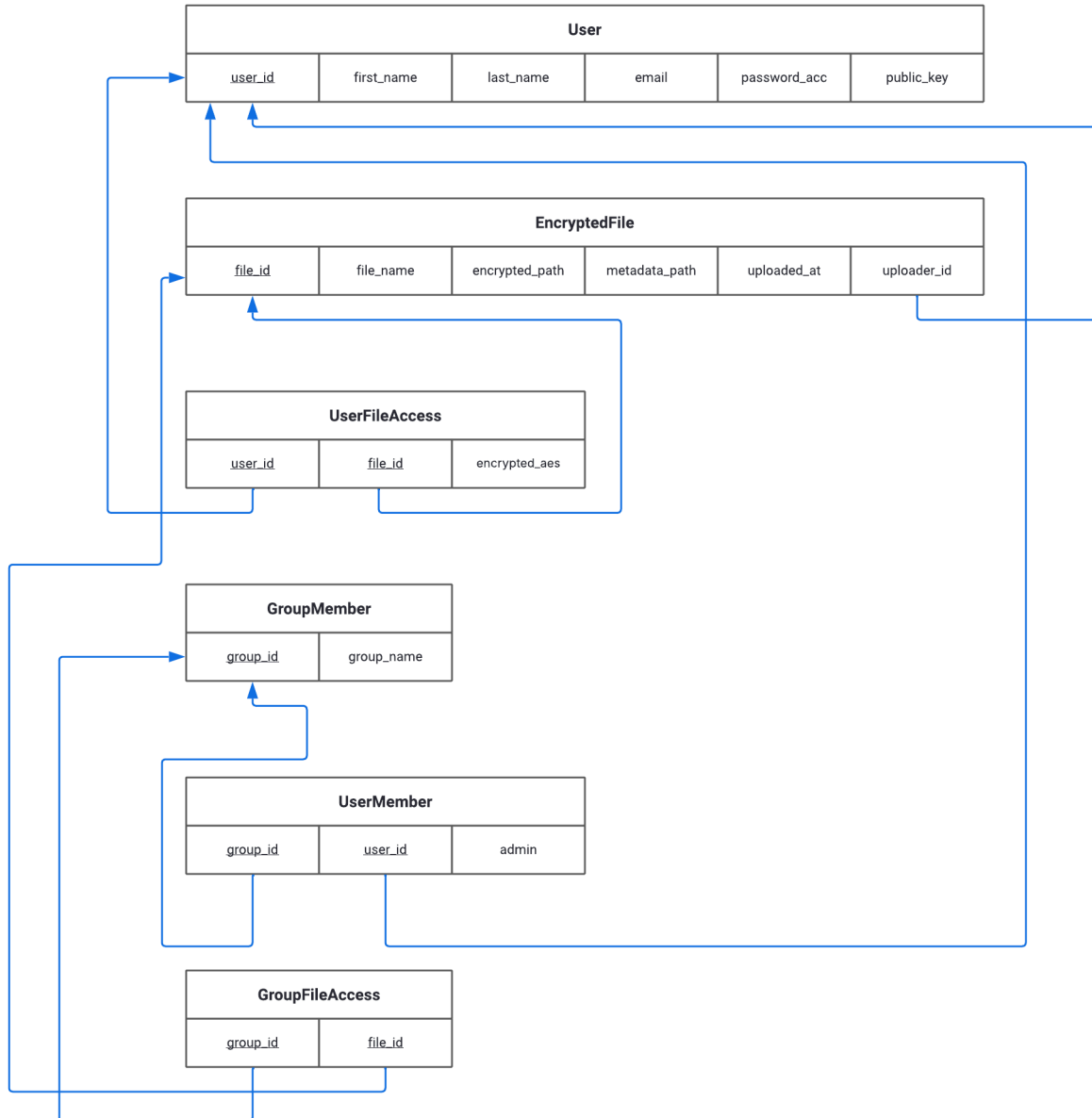
- Program structure:

```
SecureCloudStorage.sln
|
|— SecureCloudStorage.Web/
|   |— Controllers/
|   |   |— RegisterController.cs    #Control the registration for new user as well
as the generation of the user certificates
|   |   |— SigninController.cs      #Control the sign in actions
|   |   |— HomeController.cs        #Control the home page and the navigation bar
|   |   |— GroupController.cs       #Control the group-scoped activities (creating
new groups, adding or removing members, providing file access to a group, ...)
|   |   |— FileController.cs        #Manage file-processing tasks (uploading,
downloading, encrypting, decrypting, ...)
|   |— Models/
|   |   |— ErrorViewModel.cs
|   |   |— AdminViewModel.cs
|   |   |— EncryptedFileViewModel.cs
|   |— Views/          #User Interface
|   |   |— Home/
|   |   |   |— Index.cshtml
|   |   |   |— Privacy.cshtml
|   |   |— Files/
|   |   |   |— DisplayFiles.cshtml
|   |   |   |— Upload.cshtml
|   |   |   |— UploadSuccessfully.cshtml
|   |   |— Group/
|   |   |   |— AddGroup.cshtml
|   |   |   |— DeleteGroup.cshtml
|   |   |   |— DisplayGroup.cshtml
```

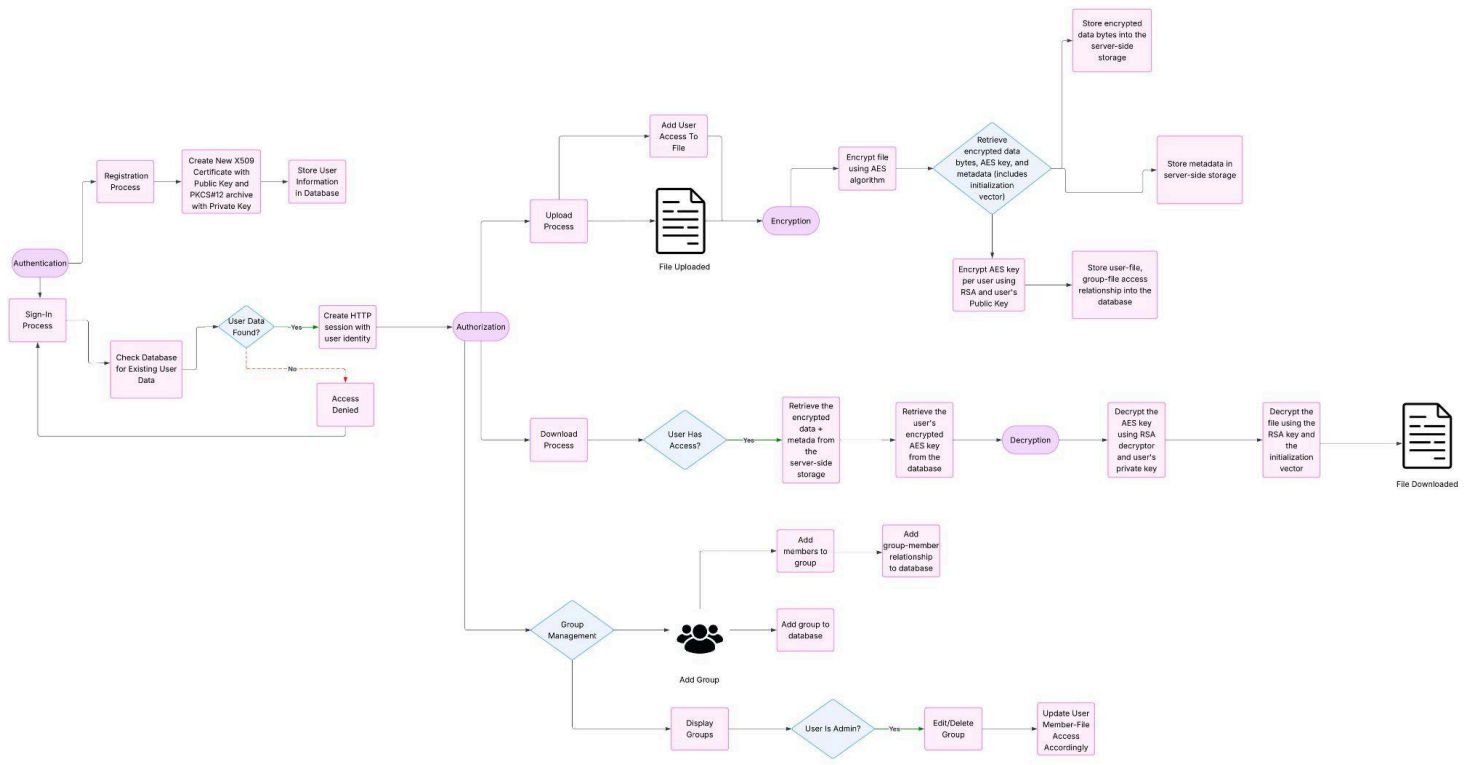
```
| | | └─ EditGroup.cshtml
| | └─ Register/
| | | └─ Register.cshtml
| | | └─ RegisterSuccessfully.cshtml
| | └─ Signin/
| | | └─ Signin.cshtml
| | └─ Shared/
| | | └─ _Layout.cshtml
| | | └─ _Layout.cshtml.css
| | | └─ Error.cshtml.css
| | | └─ _ValidationScriptsPartial.cshtml
| └─ wwwroot/
| └─ appsettings.json
| └─ appsettings.Development.json
| └─ Program.cs
|
└─ SecureCloudStorage.Application/
  └─ IEncryptionService.cs
  └─ EncryptionService.cs
  └─ AesKeyService.cs
  └─ CertificateGenerationService.cs
  |
└─ SecureCloudStorage.Infrastructure/ #Secured Storage Accessed Only from Server
Side
  └─ Data/
  | └─ AppDbContext.cs
  └─ Storage/
  | └─ uploads/ #Store encrypted version of the uploaded file
  | └─ certs/ #Store users' certificates + public keys + metadata
  | └─ certs-private/ #Store users' password-secured private keys
  | └─ metadata/ #Store users' password-secured private keys
  └─ CertificateGenerationService.cs
  |
└─ SecureCloudStorage.Domain/
  └─ Entities/ #Binding with the MySQL Database
  | └─ User.cs #Binding with the User table (storing users' information)
  | └─ EncryptedFile.cs #Binding with the EncryptedFile table (storing
encrypted uploaded files' information - file name, uploader, time)
  | └─ UserFileAccess.cs
  | └─ Group.cs #Binding with the GroupMember table (storing groups'
information)
  | └─ GroupMember.cs #Binding with the UserMember table (linking users with
their groups and indicating whether a user has admin right or not)
  | └─ GroupFileAccess.cs #Binding with the GroupFileAccess table (linking
groups with the files that all members can access)
```

```
|— FileMetadata.cs
|— UserCertificate.cs
```

- Database Structure:



- Flow Diagram:



- GitHub: <https://github.com/FayNguyen03/secure-cloud-storage-application.git>

- Demo video:  Project2.mp4