Data structures:

- Struct user:
    - HMAC(Argon2Hash(password,salt)) - for verifying correct user
    - Salt1 and Salt2
    - SymEnc(Storage) - key is Argon2Hash(password and salt 2)
- Struct Storage: Used to store what files the user has
    - Dictionary - dictionary of all fileNames and their corresponding UUIDs and private keys. If the file was a shared one, it will also contain the sender's username
    - HMAC(Argon2Hash(password,salt)) - for verifying integrity
    - Private key - private key of this user used for public key decryption
- Struct File:
    - contentUUID - uuid that points to the content structs
    - verifierUUID - uuid that is used to verify the integrity of the content structs
    - prevUUID - uuid that points to the previous node of the struct files
    - sharedDictionary - dictionary of users and who they shared the file to
    - invDictionary - dictionary of users and their invitation struct UUIDs
    - HMAC(creator username) - for checking if user has right to revoke
- Struct Content:
    - Content - piece of the actual content of the file
    - verifierUUID - uuid that is used to verify the integrity of the content structs
- Struct Invitation:
    - Creator name - username of the invitation's creator
    - Recipient name - username of the recipient's
    - fileUUID - UUID of the file
    - Key - private key to decrypt the file

Helper functions:

- ContentAssemble(filename string) - helps reassemble the file content in LoadFile.
- ReassignAccess(filename String, user String) - maintain user access to a shared file after a different user has been revoked
- DelInvitation(filename String, users List) - delete the invitations granted to the users in list

User Authentication:

When first initializing a user, we take in the username and password and make sure the username is valid. Generate a salt and then use Argon2Hash to slow hash the password along with the salt, and hmac the result after. Create a User Struct and add the salt to it along with the hashed password. Generate a second salt (salt2) and store it in the struct. Create a storage struct and add the hashed password into it. Generate a PKE key pair and store the public key in the keystore, and the private key in the storage struct. Encrypt using Argon2Hash(password, salt2) as key and add it to the user struct. Marshal this struct and add it to the Datastore with the UUID of the username. Then store When the user tries to login, look for the uuid that corresponds to the input username and verify that the passwords match using the salt and hashed password.

Multiple Devices Synchronization:

Make sure whenever there is a file update, it is always updated on the DataStore. Whenever a user calls LoadFile, make sure it is retrieved from the DataStore with the latest updates.

File Storage and Retrieval:

For FileStore, create a File struct. Generate three random UUIDs contentUUID, verifierUUID, and fileUUID. Create a content Struct and store the content as well as the verifierUUID in there.

SymEncrypt the content struct using a randomly generated private key as the private key. DataStoreSet the encrypted content with the ptrUUID. Store ptrUUID and verifierUUID into the File Struct. Encrypt the file using SysEnc and the same private key. Using the fileUUID, DataStoreSet the encrypted file. Store the fileUUID inside the dictionary of the user's storage struct.

For LoadFile, get the fileUUID from the storage struct. DataStoreGet the corresponding file. Decrypt the file using the private key. While prev Ptr of the File Struct is not null, use the ptrUUID stored in the File Structs and decrypt the appended data bytes. Iterate backwards until the entire file content has been reconstructed. This file structure is similar to a linked list, except the data contents are stored in reverse order. If the file the user is trying to access is a file they got access through an invitation, their storage struct dictionary will have the uuid of the invitation. Access the invitation and use the information in that to gain access to the file.

Efficient Append:
Each time AppendToFile is called, a new File Struct (A) and its Content Struct (C_A) is created. The File Struct that the fileUUID used to point to (B) is moved to a new UUID and the fileUUID points to A. A contains a pointer to C_A and its prevUUID points to B. Therefore the total bandwidth of the call to append is the size of the content + the constant cost of calling DataStoreSet and DataStoreGet on B.

File Sharing:
Generate a random UUID called randUUID. Create an Invitation Struct and store the UUID of the file you want to share. Store the key of the file in the Invitation Struct along with the creator and recipient's username. PublicEnc the struct using the recipient's public key. DataStoreSet the randUUID and the encrypted invitation struct.

For acceptInvitation, decrypt using the private key to get the Invitation Struct. Use the fileUUID stored in the struct invitation to retrieve the file and use the stored private key to decrypt it. Add the creator's name to the sharedDictionary and set the recipient's name to its value. Add the recipient's name to the invDictionary and set its value to the invitation UUID. In the user storage struct, add the fileName and set its value to the fileUUID and the creator username.

File Revocation:
Retrieve the file first and the list of users the revokedUser shared to. Go through each of those users and delete their corresponding invitations using the invDictionary and also delete those values from the invDictionary. Generate a new random UUID and assign that as the new pointer for the file, updating the value in the creator's storage struct as well. Going through all the other users that the file has been shared with, access their invitations and update the UUIDs of the file location. Generate a new private key for the file and update them in the invitation structs.