Users And User Authentication

In this section, you'll design two constructors to support creating new users and letting users log in to the system.

EXAMPLE

This example scenario illustrates how to create new users and let existing users log in.

- EvanBot calls `InitUser("evanbot", "password123")`.
  - This creates a new user with username "evanbot" and password "password123". If the username "evanbot" already exists, the function would return an error.
  - This constructor function creates and returns a User object with instance variables. EvanBot can call the instance methods of this object to perform file operations.
- There is no log out operation. If EvanBot is done running file operations, they can simply quit the program, which will destroy the User object (and its instance variables). This should not cause any data to be lost.
- Later, EvanBot runs your code again and calls `GetUser("evanbot", "password123")`.
  - This constructor function should create and return a User object corresponding to the existing EvanBot user. As before, the object can have instance variables, and EvanBot can call the instance methods to perform file operations.
  - If the password is incorrect, the function would return an error.
- CodaBot calls `InitUser("codabot", "password123")`.
  - The function should create and return a User object corresponding to the new CodaBot user.
  - Note that different users could choose the same password.

InitUser

```
InitUser(username string, password string) (userdataptr *User, err error)
```

Constructor of the `User` class, used when a user logs in for the first time.

Creates a new User object and returns a Go memory pointer to the new User struct. The User struct can include any instance variables for the corresponding User object.

Recall that when the program quits, the User struct and all its data will be lost. Only data on Datastore and Keystore persists after the program quits.

Returns an error if:

1. A user with the same username exists.
2. An empty username is provided.

GetUser

```
GetUser(username string, password string) (userdataptr *User, err error)
```

Constructor of the `User` class, used when an existing user logs in.

Creates a User object for a user who has already been initialized with `InitUser`, and returns a Go memory pointer to it.

If you stored the data of the User struct to Datastore in `InitUser`, then you could download that data from Datastore and create a new User object in local memory using that data.

Returns an error if:

1. There is no initialized user for the given username.
2. The user credentials are invalid.
3. The User struct cannot be obtained due to malicious action, or the integrity of the user struct has been compromised.

DESIGN QUESTION

Design Question: User Authentication: How will you authenticate users?

How does EvanBot create a new user? When they try to log in, how will you check if the username and password are correct? How do you ensure that a user cannot login with an incorrect password?

Design Requirements: Usernames and Passwords

Usernames:

- Each user has a unique username.
- Usernames are case-sensitive: `Bob` and `bob` are different users.
- Usernames can be any string with 1 or more characters (not necessarily alphanumeric).

Passwords:

- Different users might choose to use the same password.
- The passwords provided by users have sufficient entropy for the PBKDF slow hash function to output an unpredictable string that an attacker cannot guess by brute force.
- The passwords provided by users do not have sufficient entropy to resist brute-force attacks on any of the other fast hash functions (`Hash`, `HashKDF`, or `HMAC`).
- Passwords can be any string with 0 or more characters (not necessarily alphanumeric, and could be the empty string).

Design Requirements: Multiple Devices

Users must be able to create multiple User instances on different devices. In other words, a user should be able to call `GetUser` multiple times, with the same username and password, to obtain multiple different copies of the `User` struct on multiple different devices.

All changes to files made from one device must be reflected on all other devices immediately (i.e. a user should not have to call `GetUser` again to see the changes).

EXAMPLE

This example scenario illustrates how users should be able to create multiple User instances on multiple devices:

- EvanBot has a copy of the system's code running on their laptop. EvanBot has another, duplicate copy of the system's code running on their phone.
- On the laptop, EvanBot calls `GetUser("evanbot", "password")`.
  - The system creates a User object in the laptop's local memory. We'll denote this object as `evanbot-laptop`.
- Without terminating the code running on the laptop, EvanBot calls `GetUser("evanbot", "password")` on their phone.

    - The system creates another User object in the phone's local memory. We'll denote this object as `evanbot-phone`.
    - `evanbot-laptop` and `evanbot-phone` are two different User structs. They exist on two different devices, and they both correspond to the same user (EvanBot).
- On the laptop, EvanBot calls `evanbot-laptop.StoreFile("toppings.txt", "syrup")`.
- On the phone, EvanBot calls `evanbot-phone.LoadFile("toppings.txt")` and sees "syrup".
    - Note that duplicate user objects, running on separate devices, should be able to see the latest updates to files.
- On the phone, EvanBot calls `evanbot-phone.AppendToFile("toppings.txt", "and butter")`.
- On the laptop, EvanBot calls `evanbot-laptop.LoadFile("toppings.txt")` and sees "syrup and butter".
    - It would be incorrect behavior if the system returned "syrup", because this means the append from the other device was not properly synced.

DESIGN QUESTION

Design Question: Multiple Devices: How will you ensure that multiple User objects for the same user always see the latest changes reflected?

EvanBot logs in on their laptop and phone, creating two User objects. If EvanBot makes a change on their laptop (e.g. storing a file), how do you ensure that EvanBot will see the change reflected on their phone?
