Backend Assignment - Virtual File System Implementation

Objective

Implement a virtual file system with user and file management capabilities using GoLang 1.20+.

Assignment Details

Your task is to create a virtual file system as an REPL (Read-Eval-Print Loop) command line program.

When the program starts, it should prompt the user to enter commands to interact with the virtual file system, and the program should respond accordingly, displaying the results of the commands, then prompt the user for the next command.

There is no need for container, environment variable, or client/server communication, just a simple command line program that can be run on a local machine.

There is no need for a persistent storage mechanism. The program should store all data in memory and reset the data when the program restarts.

This system should have the following capabilities:

1. **User Management:**
    - Allow users to register a unique, case insensitive username.
    - Users can have an arbitrary number of folders and files.
2. **Folder Management:**
    - Users can create, delete, and rename folders.
    - Folder names must be unique within the user's scope and are case insensitive.
    - Folders have an optional description field.
3. **File Management:**
    - Users can create, delete, and list all files within a specified folder.
    - File names must be unique within the same folder and are case insensitive.
    - Files have an optional description field.

Technical Requirements

1. The program must be implemented using GoLang 1.20+.
2. Include unit tests for the implemented features.
3. Develop the repository using Git, and follow best practices and conventions for committing code.

Your repository should contain the following components:

- **README.md:** Follow the GitHub convention for composing the README.md file, detailing instructions for setup and usage.
- **Source Code:** All source code files for the implementation.

- **Unit Test Files:** All files containing unit tests for the implemented features.

Evaluation Criteria

Your assignment will be evaluated based on the following structure: (applicable to both Junior and Senior roles, with a total score of 100 points. The passing score for Juniors is 60 and for Seniors is 75.)

1. **Correctness (15 points)**
    - We will execute your program using examples from the given specification to perform basic validation.
2. **Attention to Details (15 points)**
    - We will conduct exploratory testing to assess your ability to handle unexpected operations or edge cases effectively.
3. **Code Quality (15 points)**
    - Your project should adhere to the standard Go language conventions, ensuring readability and maintainability.
4. **Architecture Design (15 points)**
    - The structure and code of your project should clearly convey your architectural design principles.
    - Explanations should be provided in the README file if necessary. We firmly believe that a good engineer should be able to explain their design decisions.
5. **Unit Testing (15 points)**
    - Your project must include a sufficient amount of unit tests to ensure code quality.
    - These tests should follow basic unit testing principles.
    - Evaluation will include but not limit to code coverage. 90% of code coverage will make good impression, but it doesn't mean 60% is bad.
    - You are encouraged to communicate your testing strategy in your code, inline documentation, or the README file.
6. **Advanced Testing Techniques (15 points)**
    - On top of basic unit testing principles, advanced testing techniques may be necessary.
    - We will evaluate your familiarity with these techniques.
7. **Git Usage (5 points)**
    - We will review your Git Log History to understand your implementation order and thought process.
8. **README Quality (5 points)**
    - This includes the clarity of instructions and the overall presentation of the README document.

Command Specification

Your program should support the following commands:

Note:

- All the messages of successful or warning command executions should output to STDOUT.
- All the Error messages should output to STDERR.
- The token surrounded by [...] is a user input/variable.
- The question mark(?) within the [...]? indicates that token/user input is optional.

1. User Registration

- `register [username]`

  Response:

    - Add `[username]` successfully.
    - Error: The `[username]` has already existed.
    - Error: The `[username]` contain invalid chars.

2. Folder Management

- `create-folder [username] [foldername] [description]?`

  Response:

    - Create `[foldername]` successfully.
    - Error: The `[username]` doesn't exist.
    - Error: The `[foldername]` contain invalid chars.
    - Error: The `[foldername]` has already existed.

- `delete-folder [username] [foldername]`

  Response:

    - Delete `[foldername]` successfully.
    - Error: The `[username]` doesn't exist.
    - Error: The `[foldername]` doesn't exist.

- `list-folders [username] [--sort-name|--sort-created] [asc|desc]`

  Response:

    - List all the folders within the `[username]` scope in following formats: `[foldername]` `[description]` `[created at]` `[username]`

      Each field should be separated by whitespace or tab characters. The `[created at]` is a human-readable date/time format.

      The order of printed folder information is determined by the `--sort-name` or `--sort-created` combined with `asc` or `desc` flags.

      The `--sort-name` flag means sorting by `[foldername]`.

      If neither `--sort-name` nor `--sort-created` is provided, sort the list by `[foldername]` in ascending order.

    - Warning: The `[username]` doesn't have any folders.

    - Error: The `[username]` doesn't exist.

    - Prompt the user the usage of the command if there is an invalid flag.(should output to STDERR)

- `rename-folder [username] [foldername] [new-folder-name]`

  Response:

    - Rename `[foldername]` to `[new-folder-name]` successfully.
    - Error: The `[username]` doesn't exist.
    - Error: The `[foldername]` doesn't exist.

3. File Management

- `create-file [username] [foldername] [filename] [description]?`

  Response:

  - Create `[filename]` in `[username]`/`[foldername]` successfully.
  - Error: The `[username]` doesn't exist.
  - Error: The `[foldername]` doesn't exist.
  - Error: The `[filename]` contains invalid chars.
  - Error: The `[filename]` has already existed.

- `delete-file [username] [foldername] [filename]`

  Response:

  - Delete `[filename]` in `[username]`/`[foldername]` successfully.
  - Error: The `[username]` doesn't exist.
  - Error: The `[foldername]` doesn't exist.
  - Error: The `[filename]` doesn't exist.

- `list-files [username] [foldername] [--sort-name|--sort-created] [asc|desc]`

  Response:

  - List files with the following fields: `[filename]` `[description]` `[created at]` `[foldername]` `[username]`

    Each field should be separated by whitespace or tab characters.

    The `[created at]` is a human-readable date/time format.

    The order of printed file information is determined by the `--sort-name` or `--sort-created` combined with `asc` or `desc` flags.

    The `--sort-name` means sorting by `[filename]`.

    If neither `--sort-name` nor `--sort-created` is provided, sort the list by `[filename]` in ascending order.

  - Warning: The folder is empty.

  - Error: The `[username]` doesn't exist.

  - Error: The `[foldername]` doesn't exist.

  - Prompt the user the usage of the command if there is an invalid flag.(should output to STDERR)

Input Validation and Restrictions

The program is expected to perform input validation. If the command is not recognized by the program, it should notify the user that it is an invalid command or display the usage of the program. To simplify the implementation, it is acceptable for the `[username]`, `[foldername]`, `[filename]`, and `[new-folder-name]` not to contain whitespace characters.

As part of the input validation process, you should also define and enforce user input restrictions for usernames, folder names, and file names, such as maximum input length and valid character sets. Please ensure that these restrictions are clearly documented in your README.md file. Consider real-world constraints and potential issues when defining these restrictions, keeping in mind the usability of the virtual file system and the need to prevent issues such as excessively long inputs or invalid characters.

[BONUS] If you wish to challenge yourself, you can choose to implement a more advanced version that allows these tokens to have whitespace characters. In that case, all tokens with whitespace characters should be enclosed in double quotes, e.g., `"New Folder"`. Ensure that the program can handle such inputs correctly.

Example

The "#" below is a prompt to inform the user that they can type commands. The following examples demonstrate the usage of various commands in the virtual file system:

Register two users, user1 and user2

```
# register user1
Add user1 successfully.
```

Register user2

```
# register user2
Add user2 successfully.
```

Create a folder for user1 and user2 with the same folder name

```
# create-folder user1 folder1
Create folder1 successfully.

# create-folder user2 folder1
Create folder1 successfully.
```

Attempt to create a folder with an existing name for user1

```
# create-folder user1 folder1
Error: The folder1 has already existed.
```

Create a folder with a description for user1

```
# create-folder user1 folder2 this-is-folder-2
Create folder2 successfully.
```

List folders for user1 sorted by name in ascending order

```
# list-folders user1 --sort-name asc
folder1 2023-01-01 15:00:00 user1
folder2 this-is-folder-2 2023-01-01 15:00:10 user1
```

List folders for user2

```
# list-folders user2
folder1 2023-01-01 15:05:00 user2
```

Create a file with a description for user1 in folder1

```
# create-file user1 folder1 file1 this-is-file1
Create file1 in user1/folder1 successfully.
```

Create a file named config with a description for user1 in folder1

```
# create-file user1 folder1 config a-config-file
Create config in user1/folder1 successfully.
```

Attempt to create an existing file.

```
# create-file user1 folder1 config a-config-file
Error: the config has already existed.
```

Attempt to create an file for an unregistered user.

```
# create-file user-abc folder-abc config a-config-file
Error: The user-abc doesn't exist.
```

Attempt to type a unsupported command

```
# list data
Error: Unrecognized command
```

Attempt to list files with incorrect flags

```
# list-files user1 folder1 --sort a
Usage: list files [username] [foldername] [--sort-name|--sort-created]
[asc|desc]
```

```
# list-files user1 folder1 --sort-name desc
file1 this-is-file1 2023-01-01 15:00:20 folder1 user1
config a-config-file 2023-01-01 15:00:30 folder1 user1
```

Submission

To submit your completed assignment, follow these steps:

1. Ensure that your repository contains all the required components, including the Git commit history.
2. Pack/zip the entire repository. Make sure the zipped file contains the `.git` folder to preserve the commit history.
3. Attach the zipped file to an email and reply to the email containing the assignment details.

Please complete the assignment within one week of receiving the email containing the assignment details.

Please take your time to ensure that your submission meets the requirements and is of high quality, and if you require extra time or need to postpone the submission, kindly inform us in advance by replying to the email.