Implementing a Shell in Go

Introduction

In this project, you will design and implement a **shell** using the **Go programming language**. The shell should continuously accept user commands, execute them, and display the appropriate output. The goal of this project is to create a functional, efficient, and user-friendly shell that **complies with** POSIX standards while exploring various use cases.

This project will focus on:

- **Basic Shell Commands**: Handling basic commands and built-in types.
- **Executables**: Support running system executables.
- **Error Handling**: Providing clear and meaningful error messages for invalid commands or incorrect syntax.
- **User Management**: Managing users registration, login and logout.
- **Input & Output Redirection**: Supporting file redirection (`>` , `<`).
- **Database Basic Knowledge & Query**: Implementing analytical queries on previous commands and using database for data persistence.

Part One: Basic Commands

In this section, you'll implement the following commands in your shell:

1. `exit` Command

This command exits the shell and stop the program. it can be used with or without a status code.

- **Without status code:**

```
$ exit
exit status 0
```

  This exits the shell with the default status code 0.

- **With status code:**

```
$ exit 2
exit status 2
$ exit 1 2
too many arguments
```

  This exits the shell with the specified status code (in this case, 1).

2. `echo` Command

This command prints the provided arguments to the output. it should support environment variables, multiple expressions and literal values.

- **Examples:**

```
$ echo hello
hello
```

```
$ echo no$PATH
no/usr/local/bin:/usr/bin:/bin
```

```
$ echo no yes
no yes
```

```
$ echo 'no$PATH'
no$PATH
```

3. `cat` Command

This command displays the content of the specified file.

- **Example:**

```
$ cat filename.txt

```

```
(contents of filename.txt)
```

4. `type` Command

This command indicates whether a command is a shell builtin or an executable in the system. it is shell builtin if it has been implemented in your program (e.g., `cd`, `echo`, `cat`, etc.). if it's not shell builtin, you have to print the path of the executable. for executables, first you have to fetch the $PATH value, then search among files in each directory in the path and if the name found, return the address of it.

- **Examples:** My Path: /usr/sbin:/usr/bin Files in /usr/bin: git nslookup ...

```
$ type cd
cd is a shell builtin
```

```

$ type git
git is /usr/bin/git
```

5. Executing System Commands

The shell should be capable of running system executables.

- **Example:**

```
$ git status
```

This should execute `git status` in the current working directory.

6. `pwd` Command

This command prints the current working directory.

- **Example:**

```
$ pwd
/home/user/current_directory
```

7. `cd` Command

This command changes the current working directory.

- **Example:**

```
cd /home/user/new_directory
```

After executing this command, the working directory changes to `/home/user/new_directory` .

8. ` Handling Unknown Commands

If the user enters a command that the shell or system doesn't recognize, an appropriate error message should be displayed.

- **Example:**

```
$ unknowncommand
unknowncommand: command not found
```

9. User Management

Users should be able to login with this command:

```
$ login {username} {password | empty}

$ adduser {username} {password | empty}

{username}:$ logout
```

- **Examples:**

```
$ adduser hosseint 1234
user created successfully
```

```
$ adduser hosseint
duplicate user exists with this username
```

```
$ login hosseint 1234
hosseint:$
```

```
hosseint:$ logout
$
```

10. History

This shell should support history of executed commands for each user. The result should be in descending order of count, and among commands with the same count, the newest timestamp should come first.

- **Example:**

```
$ ls
$ ls
$ pwd
$ cat
```

history should look like this

```
$ history
| ls  | 2 |
| cat | 1 |
| pwd | 1 |
```

- When the user is **not logged in**, they only see the commands in that session.
- When logged in, the user sees the commands persisted for that user. The history will remain intact across session restarts.
- No user can see other users' commands history, including the guest user.
- `history clean` clears the history for that user, or if executed without a user, clears the session

```
$ history clean
$ history
empty command history
```

Part Two: Bonus Features

To enhance your shell and earn extra credit, consider implementing the following features:

1. Redirections

The shell should support input and output redirections.

- **Output without appending:**

```
$ command > file
```

or

```
$ command 1> file
```

These commands redirect the output of `command` to `file`, overwriting its existing content.

- **Output with appending:**

```
$ command >> file
```

or

```
$ command 1>> file
```

These commands append the output of `command` to the end of `file`.

- **Error redirection:**

```
$ command 2> error_file
```

This command redirects the error output of `command` to `error_file`.

```
$ command 2>> error_file
```

This command appends the error output of `command` to the end of `error_file`.

2. Double Quotes

The shell should support double-quoted strings and allow escaping special characters using the backslash (`\`). The backslash retains its special meaning only when followed by one of the following characters **[$, `, ", \ ]** or `newline`. Within double quotes, backslashes that are followed by one of these characters are removed.

- **Example:**

```
$ echo "This is a \"quoted\" word."
This is a "quoted" word.
```

In this example, the backslash before the double quote prevents it from being interpreted as the end of the string.

```
$ echo "ab \a \$ \` \" \\"
ab \a $ ` " \
```

Suggested Resources

To better understand the concepts and implement this project, consider the following resources:

- Bash Reference Manual

  This resource provides guidance and examples for implementing a shell.

Final Notes

- **Code Structure:** Aim to write modular and readable code to facilitate maintenance and future development.
- **Error Handling:** Ensure proper error handling in your command implementations and display meaningful error messages.
- **Testing and Evaluation:** Write unit tests thoroughly to cover all scenarios and ensure correct functionality.

By completing this project, you'll gain a deeper understanding of Unix shell concepts and system programming with Go. Good luck!