CT213 – Assignment 1 Report

BashBook V1.0

Report and Project by Alasdair and Tim

Introduction

As the assignment brief lays out, this project implements a simple Facebook-like social media server using Bash. The project successfully implements the following five scripts: Creating a User, Adding a Friend, Posting Messages on User Walls, Displaying Walls and the Server Script. Data Storage is done through the creation and manipulation of text directories in the Bashbook database directory on the desktop. Once a user is created, they are given their own directory in which their wall and friends are stored. Bash scripts are run in the Linux terminal which are used to modify user's walls and add friends. All outputs for the scripts follow the format provided in the brief. All five scripts also contain basic error checks, which are used to ensure that the arguments entered by the user are valid. We believe this project satisfies all of the criteria laid out in brief and it meets the standards for a functioning and stable V1.1 of BashBook.

System Organisation

BashBook revolves mainly around the manipulation of directories and associated text files; almost every script implemented contains a '-d' directory check, which ensures that the directory exists on a user's machine. Usually if the directory exists, a text file within that directory is edited. The text file is edited based on the arguments entered by the user when the server script file is run. The 'create.sh' script revolves around the creation of files and directories using Bash. The script creates a directory within the Bashbook general directory for the user once they create their account. Within the user directory, the wall.txt and friends.txt files are made. In the 'add_friend.sh' script file, the user decides which friend they want to add to their account (i.e., which friend to add to their text file in their associated user directory.) The 'display_wall.sh' script reads and prints the wall.txt file in whichever person is specified by the user on the terminal. Many functions also read in text files to ensure that data isn't repeated such as add friend which must check that a friend of the same name has not already been added.

The project also implements a basic 'sender-receiver' concept. This concept is used in the 'post_message.sh' and 'add_friend.sh' scripts which both require two user ids. The sender is the one who is editing the text file of the receiver. In 'post_message.sh', the sender enters a string and the walls.txt file of the receiver is amended with that string. The name of the sender, and the message is added to the wall.txt file so the receiver will know which user posted the message to their wall. 'add_friend.sh' also employs this concept, the first argument in this case is the receiver and the second argument is the sender. The user's friends.txt file receives the name of the friend, and is appended to the file (i.e., the two users are now friends). As mentioned before, checks are implemented to ensure that all arguments entered are valid in order to ensure that a text file of a user that does not exist can be edited.

The organisation of 'server.sh'. As mentioned in the brief, this script creates an infinite loop which processes commands to be used in Bashbook. Our main addition was the 'read' command, which ensures that the arguments are correctly read from the terminal. The first input corresponds to the request, which will specify which script will run (for example, create will run the create.sh script). Each parameter after the request will be passed into the necessary scripts. Each '#do something' is

replaced with a simple command to run each individual script. The majority of the organisation for this section of the project is given in the brief.

Lastly, I want to give a brief summary of the interaction between the different scripts. The core of this interaction is based around the 'server.sh' script. This is the script which acts as the 'glue' and allows for the running of all the other scripts. After server.sh begins running the terminal is continuously checked for commands. Any of the commands can then be run by typing a simple keyword into the terminal as opposed to the whole command which would normally be required. The interaction between the other 4 scripts is limited and mainly revolves around making sure that any text added to a text file is formatted correctly so that it can be correctly read in by another script if needed.Say when posting a message to a wall we added the text to the file without spaces and new lines, reading in this text and printing it to terminal in display_wall would require a lot of parsing and significantly more effort. This process is greatly simplified when we ensure that the text file is edited in a formatted and logical way. File navigation is handled by implementing a directory of scripts on the desktop. Within that directory, the server.sh file is located. Upon running the server the file, a general Bashbook database directory is created on the desktop. The terminal will navigate to the desktop of the user and create it. Once the general database is created, all other script files will navigate, check and create directories inside that Bashbook database when called. The manipulation of Linux's cd command allows for precise navigation which proved extremely useful in file navigation.

Features and Implementation of Each Group Member

After we discussed the assignment brief and scripts, we brainstormed on possible implementations of each script. Both of us created our own implementations of the five scripts. We consulted each other on specific aspects. We both assisted each other when problems arose throughout the process. Once we completed our individual implementations of the scripts, we had to decide whose script we would use in the final implementation. We settled on using two scripts created by one and three from the other.

Alasdair

2.1.1 Creating a User

Creating a User was one of the simpler functions of the project. I first checked the validity of the arguments entered, then a directory was created which with the name of the string argument entered into the program. The directory was created with the 'mkdir $1 command. Using the terminal 'touch' command two files friends.txt and wall.txt were created within the new directory. These text files are empty and will serve a use for other functions later on. Different arguments are output to the terminal depending on whether the program was successfully executed, different reasons for failure print different errors to the terminal.

2.1.2 Adding a Friend

Adding a friend was considerably more difficult than the 'creating a user' function. This function uses a sender and receiver format which was mentioned above. The first argument is the receiver in this case and the second is the sender. The friends.txt file of the receiver is amended to add the name of the sender. Before this can be done the friends.txt file of the receiver must be checked to ensure that it does not already include the name of the sender. To do this I had to read in the file and compare it to the sender. I did this by using 'if grep' our output is thrown into /dev/null file as I only need to

know if the file contains a specific string I do not need to store the actual content of the file. By far the most time-consuming aspect of this feature was implementing the check having a friend already added.

Tim

2.1.3 Posting Messages (post_messages.sh)

At the start of my script, I've decided to implement an if statement to check to see if the number of arguments passed into the script is greater than three. The first two inputs are sender and receiver ids that will be used to validate their friendship and eventually post the message on the receiver's wall. The third input and everything beyond will be the message string that the sender would like to post.

If statements are used to check if both the sender and receiver exist as users on the Bashbook server. They implement the '-d' option within the if statement to check if their respective file directory paths exist on the server. If all checks are passed, grep is used to check if the sender's name is listed within the receiver's friends.txt text file (i.e., check to see if the sender and receiver are friends). If the sender is not listed, inform the user that the sender needs to be friends with the receiver to post messages.

If the sender's name is listed on the receiver's text file, two shift commands are used to shift all the arguments to the left. This will make the beginning of the message as the first argument in the script. Then all the arguments remaining after the double shift are assigned to a variable using '$@'. Using echo, this message variable is posted to the user's wall text file.

2.1.4 Displaying Walls (display_wall.sh)

I've used an if statement to check to see if there are no arguments passed into the function. The user is formed if that is the case with a console message. If it passes this test, a variable is assigned the first argument of the script.

An if statement is used to check the directory path, which is a standard path with the user id passed into the script concatenated on the end. If the file path exists, the user's wall text file is printed to the console screen. Statements are printed before and at the end of the printed message to inform the user of when a message begins and when a message ends.

2.1.5 Server Script (server.sh)

At the beginning of the script, the user is welcomed to Bashbook. The Bashbook general directory is created soon after. The script then enters a while loop. The console then asks the user to enter a request and the necessary arguments. The available commands and arguments are as follows:

create (userid)

add (userid) (friendid)

post (senderid) (receiverid)

display (userid)

The loop enters a switch case and parses the request type. If the request does not match the four commands available, the user is informed of the available and the program exits. Upon matching a request, if statements are put in place to check for empty required arguments for scripts. If they pass the check, the associated script is run along with the arguments

Challenges **Encountered**

Alasdair

The majority of the challenges that I faced when implementing my scripts were syntax related. I wasted a lot of time before realising that the exact correct type of quotes were needed in order for bash to recognise terminal commands. Using fi and done was also challenging as at times it was unclear which if statement was connected to each fi. This became especially frustrating when using multiple if statements or if statements inside other statements. This also applied to the while condition. I also found on multiple instances that even though my program worked it would still print an error to the terminal. The most common of these was 'typeset not found' which would print when I was testing the program before proceeding to correctly execute the program. This made the output look soppy and eventually I had to remove this from my implementation. All of these errors applied to both create.sh and add_friend.sh.

One other major issue I had was in add_friend when I had to check whether a friend already existed. I began by using the cat command but as that reads in the whole file as one it was impossible to use for a comparison and parsing seemed more difficult than it was worth.. Another minor issue was the different keyboard on the VM. This lead to many instances of wasted time where I had to search up specific ascii symbols online and copy paste them into my code as I couldn't figure out how to type them on the different VM keyboards one example of this was the or(||) operator.

The last major hurdle I faced was in the function post_Messages, when taking in the message as an argument I had trouble parsing the message from the rest of the arguments and ensuring that the message was correctly stored in a variable. Despite this hurdle I ended up learning a lot about the interaction between the bash and the Linux terminal and how bash processes commands from Linux. Issues with processing terminal arguments also presented themselves in the 'server.sh' function. Where each program had to take a different number of arguments

Tim

The biggest problem I faced during this assignment was file navigation. I encountered this problem across all of the scripts that I wrote, especially in 'add_friend.sh'. I had problems creating file paths which were used to navigate to the correct areas in the Bashbook directory. Through the use of the cd command, I was able to properly navigate to the correct areas of the Bashbook database and create necessary users, check file paths and append files.

Within the 'post_messages.sh' script, I've had difficulty making the message post to the text file. The Linux terminal parses arguments based on spaces, and a message can have multiple spaces. I've eventually settled on a solution that allows me to parse the message from the arguments. Once the first two inputs have been assigned to variables, I shifted the arguments twice. Then I assigned "$@" to the message variable. This allowed me to parse the entire message including the spaces into a variable and post it to the receiver's wall.

During the creation of the 'server.sh' script file, I had difficulties checking if particular variables were empty or not. I needed to check if they were empty in order to implement a "nok: Bad Request"

output. The output was generated whenever one of the required arguments for any of the scripts requested were not present. I overcame this problem by implementing an if statement that checks if the variables needed were parsed correctly from the terminal. Through the use of the OR operator, I was able to check multiple variables in a single if statement.

Within the ‘create.sh’ script file, it took me a while to wrap my head around ‘fi’ statements at the end of if statements. I continued to practice using if statements until I understood them which proved useful in later scripts, where more if statements were required.

The ‘display_wall.sh’ script proved to be the least troublesome. I attempted to implement many other print commands, such as less and od, but the cat command proved to be the most efficient.

Conclusion

Overall we feel as though the project was a success. We met all the criteria laid out in the brief and our program can handle errors efficiently. Due to the uneven amount of script files stated in the brief, one of us had an extra script file in the final submission, but extra work on the PDF report helped to properly disperse the workload. As all of these scripts had to work in tandem with one another, it was important to ensure that the manipulation of files remained consistent throughout the program, thus our program problem and planning skills improved considerably. This was especially true when implementing ‘server.sh’. We were aware that one script may have to append one text file and another script may have to read and append to it, without consistency many errors could arise in the manipulation of data. Planning, collaboration and getting the first version of Bashbook up and running took about 6 hours in total.

We believe that our Bash scripting improved immensely. We gained valuable knowledge of system organisation through file navigation and these skills will be important once work on the second version of Bashbook begins. Our teamwork skills were ameliorated and we look forward to working on the second version together.