A Web Server in Bash

2021-03-16

Table of Contents

This is a bit of a dumb project, there's all sorts of bugs and they're the horrifying kind. Shell scripts have access directly to the computer and so you can do all sorts of strange and harmful things. But this all makes me smile that it all even works.

Source code can be found at:
https://github.com/Krowemoh/bash-server

Chapter 1 - Hello, (to the weird) World!

Let's write the simplest bash script just to orient ourselves. Open up a text file and add just the following line and save it.

./test.sh

```
echo "Hello, World!"
```

Now we can run our script at the commandline by doing the following.

```
> bash test.sh
```

Here we are specifying the interpreter we want to use and the script we want to feed into the interpreter. We are going to be using bash but most shells will work the same way.
Once we run our command, we should get "Hello, World!" printed to our screen. By default echo will print to stdout, which is standard output.
./test.sh

```
echo "Hello, stdout!" >&1 # stdout
echo "Hello, stderr!" >&2 # stderr
echo "Hello, file!" > temp.txt # file
```

Here we are explicitly redirecting the output of echo to &1, &2 and a new file.
&1 is the file descriptor for standard output.
&2 is the file descriptor for standard error.
The lack of a space after the > is important when we are sending the output to file descriptors.

```
> bash test.sh
```

Now when we run this script we should get 2 lines of text printed to the screen and 1 line in a new file called temp.txt. Voila! We have a script that can send output to different outputs now.
Next, lets look at running our script without specifying the interpreter on the command line. We want to be able to send this script to someone else and have then just run it without having to know what shell interpreter they should use.

Chapter 2 - Starting off with a (#!sh)bang

When we try to execute a program in linux, the shell environment will call exec on the program. If it's a binary, then it would have the all needed information to start working so it can be loaded into memory and the system will pass control to the program. Scripts are a little different, they need an interpreter to read that can read the source code and do things based off it.
This means that we need to somehow let the system know what interpreter certain scripts are to use. This is where the hash tag bang symbol, #! comes in. At the top of a file we can add #! and this will tell the system what interpreter to use.
./test.sh

```
#!/usr/bin/bash

echo "Hello, World!"
```

We have now added a shbang to our script and we have specfied the full path to the interpreter we want the system to use.

```
> chmod +x test.sh
```

We mark our script as executable so that we can run it directly.

```
> ./test.sh
```

Magically, our script now runs without us using an interpreter explicitly!
The #! is a magic number, in hexadecimal it is 0x23 0x21, and when the script is called from the shell and passed to exec, exec knows that this a shell script asking for an interpreter to be run. In reality, that shbang is a comment in our bash program. It exists as a way for us to communicate directly to the system. We are saying use the program specified in our path, using this file as the input.

./test.sh

```
#!/usr/bin/ls

echo "Hello, World!"
```

Here we are saying use ls and give it the input of our file. In this case this will simply print out our file as if we had done "ls test.xh" at the commandline.

```
#!/usr/bin/bash
#!/usr/bin/ls

echo "Hello, World!"
```

The first line correctly tells the system what interpreter to use and then as soon as that's done our interpreter will begin parsing the file. Now when the interpreter sees the hashtags, it considers those all comments and ignores them.
This is also why the first line needs to be the shbang, otherwise we won't be able to tell the system what interpreter to use.
! There we have it, we can mark our script as executable, use the shbang to set our interpreter and run our script like a binary.
Next up, let's look at something that will make life easier later!

Chapter 3 - Running with arguments

Currently our little script just prints out a message, but it would be nice if we can pass in some arguments.

```
#!/usr/bin/bash

echo "Hello, $1"
```

We can access arguments by doing $1, $2, and so on, for the number of arguments we want to process. We can also get all the arguments passed in by doing $@.

```
> ./test.sh Nivethan
```

If we run our script passing in a name we should now be able to see it.
Before we can get started on working on our server, let's add one more utility that will make life easier for us. Logging!

Chapter 4 - I want to be a logger when I grow up

Currently we are printing stuff to standard output by default and we can write to standard error by redirecting output to &2. We can write an abstraction over this so we can have a log function that will write to standard output automatically and also only print those messages if we are running with the right debug level.
./test.sh

```
#!/usr/bin/bash
declare -r DEBUG=1

log() {
    if [ $DEBUG = 1 ]
    then
        echo "$1" >&2;
    fi
}

log "Testing..."
```

Here we have a log function and we can pass in just 1 argument. This is because we use $1 directly. We could do $@ and print all the arguments that passed into the log function.

The other thing to note is that we are using the declare syntax to set up a global variable called DEBUG that is read only. Our script shouldn't be able to change this while it is running.
Now with the set up out of the way, we can take a look at opening a socket and starting to build the core of our little http server.

Chapter 5 - The internet is made of (net)cats

Netcat is a linux utility that can bind to a socket and output what it gets to the screen. You can use netcat in both listening mode and connecting mode.

Using netcat

```
nc --listen 0.0.0.0 7999
```

We can run this at the commandline and we should see nc just hanging. It is waiting for something to be sent to 0.0.0.0 on port 7999.
0.0.0.0 is a shorthand to say we want to bind to all interfaces on this machine. By default netcat will bind to 127.0.0.1 which is the loopback interface but then it is only reachable from within the computer. To make it accesible outside, we would need to bind to the network facing interfaces.
We could specify the real IP address of the machine such as 192.168.1.101 but using 0.0.0.0 is straightforward and will work even if the machine's IP changes in the future.
We also specifiy the port we want netcat to listen to. To make the port available, we need to update our firewall.
/etc/firewalld/zones/public.xml

```
<port protocol="tcp" port="7999"/>
```

We add the above to our firewall and then we just need to reload our firewall.

```
# firewall-cmd --reload
```

Now we should be able to open our browser and navigate to the machine's IP address with the port number.

```
❯ nc --listen 0.0.0.0 7999
GET / HTTP/1.1
Host: 192.168.1.101:7999
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:86.0) Gecko/20100101 Firefox/86.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Cookie: compact_display_state=false; filter=all; session_id=1db271d4-26e8-439d-99b5-b7db1a377afc
Upgrade-Insecure-Requests: 1
```

Voila! We have netcat listening and the request that the browser made got printed on our screen. Now we have a way of opening a socket and listening to it, lets get started on working on our script!

Adding netcat into the mix

Netcat currently prints what it receives to the screen, what we really want it to do is to pass that data to us directly so we can process the request and respond to it. To do this, we will use the --sh-exec option which tells netcat to pass all of the input it receives and execute a new process specified by the flag.
./server.sh

```
#!/usr/bin/bash
declare -r DEBUG=1

log() {
    if [ $DEBUG = 1 ]
    then
        echo "$1" >&2;
```

```
    fi
}

echo "Hello, World!"
```

Now that we are starting to get to the real work, I've renamed our test script to server.sh. From now on we'll start slowly building out HTTP server!

```
> nc --listen 0.0.0.0 7999 --sh-exec "./server.sh"
```

We should now be to refresh our browser and we should see "Hello, World!" on our screen. Netcat was listening and when it received input, it passed it to server.sh which in turn printed "Hello, World!". Netcat will send whatever get's sent to standard output in the program it executed. This is why we don't see anything in the terminal window but do see something in the browser.

```
#!/usr/bin/bash
declare -r DEBUG=1

log() {
    if [ $DEBUG = 1 ]
    then
        echo "$1" >&2;
    fi
}

echo "Hello, World"
log "Hello, Error"
```

If we re-run our netcat command and navigate to the browser, this time we will see output in both the browser and the terminal!
Currently netcat ends after the first response we send. This is because netcat doesn't stay open by default.

```
> nc --listen --keep-open 0.0.0.0 7999 --sh-exec "./server.sh"
```

We add the --keep-open flag and with that we will keep netcat open for future connections.

Abstractions Galore!

We have heart of our server almost starting to beat. Let's do some clean up so that we can get our script to be functional by itself. Currently we need to manually run the netcat command and call our server.sh script.
We want our bash script to contain the netcat command so that we can run just server.sh and start our http server.
To do this we are going to create a serve function in our script that will open a socket and listenn and a process function that will send a message back.
./server.sh

```
#!/usr/bin/bash
declare -r DEBUG=1

log() {
    if [ $DEBUG = 1 ]
    then
        echo "$1" >&2;
    fi
}

serve() {
```

```
    nc --listen --keep-open 0.0.0.0 7999 --sh-exec "./server.sh process"
}

process() {
    echo "Hello, World"
    log "Hello, Error"
}

"$1"
```

We have written two functions now, we have our serve function that will run netcat and we have our process function which will send a message. We also have $1 right at the end our script. This means that by default if we execute our script, nothing will happen. This is because we want to be able to trigger the netcat and process it all in the same file.
We could split this up and it likely makes more sense that way but I like keeping it in one file as it helps me hold everything in my head.

```
> ./server.sh serve
```

We run our server and we call it passing in the argument serve. The $1 at the end of our file gets substituted out and it makes a function call to serve.
Now when we navigate to the browser, netcat will get some input, and trigger itself but this time passing process as it's argument. Process will then be called and voila! we will get Hello World printed to our screen and terminal!
Next, let's take a look at reading the requests the browser makes and responding to them!

Chapter 6 - Learning to read

Currently we don't do anything with the requests the browser is sending us, so as a first step let's get them printed to the screen.
The first thing we need to do is begin reading in the input that get's passed into our process function. Instead of doing the read directly in our process function though, let's create a new function get_request_headers(). However before we can get there we need to go over how the input is being passed through netcat and to bash.
Netcat will send the raw data to the process function, but once in the process function that data will be broken up by the delimiter that bash uses. Currently, that means multiple delimiters. In bash data can be broken up via any sort of white space, meaning tabs, spaces, and new lines all act as delimiters.

Internal Field Seperator

HTTP uses spaces new lines as the delimiter for each header piece and a blank line to signify the start of the body or end of the request. HTTP also uses spaces in the first header to set the type request, location and version.
We want out script to use a single delimiter, the new line, as the delimiter for all data.

```
#!/usr/bin/bash
declare -r CR=$'\r'
declare -r LF=$'\n'
declare -r CR_LF="${CR}${LF}"

declare -r DEBUG=1

...
process() {
    IFS=$LF

    echo "Hello, World"
    log "Hello, Error"
}
```

We are going to declare a few more global read only variables and then in our process function we set a special variable called IFS to the line feed character. This sets the internal field seperator to new line. This means that bash will now parse input along new line characters instead of using spaces.

This will become more obvious once we take a look at how we read data.

Reading 101

The first step is to call our request function in our process function.

```
...
process() {
    IFS=$'\n'

    get_request_headers

    echo "Hello, World"
    log "Hello, Error"
}
...
```

Now that we are calling our function to get request headers, we can now handle all the reads there.

```
...
get_request_headers() {
    request_headers=()

    while true
    do
        read header
        log "$header"
        if [ "$header" = $CR_LF ]
        then
            break
        fi
        request_headers=("${request_headers[@]}" "$header")
    done
}
...
```

The first thing we do in our function is initalize an empty array. Next we begin an infinite loop that will end when we reach a blank line. A blank line signifies the end of the request or it can signify the start of the body. For now let's take a blank line to mean the end of the request.
read is a bash function that will read in one unit of the standard input. In this case that means that it will read an entire line of a http request that netcat received. This is why we needed to change the IFS. HTTP mixes spaces and newlines and so we want to make sure we process an entire line before moving to the next line.
Once we have read in the header, we then concatenate the existing request_headers array with the new element we just read in. Once the while loop ends, we will have an array of http headers in a variable called request_headers.
Let's try navigating to our server through the browser.

```
n
❯ ./test.sh serve
GET / HTTP/1.1
Host: 192.168.1.101:7999
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:86.0) Gecko/20100101 Firefox/86.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
```

```
Connection: keep-alive
Cookie: compact_display_state=false; filter=all; session_id=1db271d4-26e8-439d-99b5-b7db1a377afc
Upgrade-Insecure-Requests: 1
Cache-Control: max-age=0

Hello, Error
GET /favicon.ico HTTP/1.1
Host: 192.168.1.101:7999
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:86.0) Gecko/20100101 Firefox/86.0
Accept: image/webp,*/*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Referer: http://192.168.7.41:7999/
Cookie: compact_display_state=false; filter=all; session_id=1db271d4-26e8-439d-99b5-b7db1a377afc
Cache-Control: max-age=0

Hello, Error
```

Voila! We have 2 requests from the browser logged. We have a request for / which is the default route. We also have a request for /favicon.ico.
If we didn't set the IFS, we would see GET, /favicon.ico, HTTP/1.1, and the rest of the sections on seperate lines.
Now that our little server can see our requests and has processed them we can start handling them. If we can return the requested things, we would have a very rudimentary web server!

Chapter 7 - Let's get down to (handling) business

Just like how we read in the headers, we are going to add another function to handle requests in our process function.

```
...
process() {
    IFS=$LF

    get_request_headers

    handle_requested_resource

    echo "Hello, World"
    log "Hello, Error"
}
...
```

We are going to handle a request by checking to see if the location requested is a valid file, and if it is we want to return the data contained in that file to the browser.
./app/index.html

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <title>Fun!</title>
    </head>
    <body>
        <h1>Hello, World!</h1>
```

```
    </body>
</html>
```

Here we have a very simple index.html in a folder called app.

```
handle_requested_resource() {
    regexp=".* (.*) HTTP"
    [[ "${request_headers[0]}" =~ $regexp ]]

    resource="${BASH_REMATCH[1]}"

    requested_resource="./app$resource"
    if [ -f "$requested_resource" ]
    then
        cat "$requested_resource"
    fi
}
```

The first thing we need to do is get the location the request is specifiying. We can use regex and we specifiy a capture group. The capture group is signfied by the (). We can then run this regex against some text or variable by enclosing it in double square brackes and using =~. This is a strange syntax. Square brackets in bash mean to test and so this is likely testing the regex against the variable, but its not easy to know just by looking at it.
The captures are then stored in a special bash variable called BASH_REMATCH. This is an array of matches and in our case it will be just an array of 1. We now have the resource being requested. We then append it to the ./app as we want to serve only the files located in app.
Next we check to see if the requested resource is a file. If it is a file we will cat the file. The cat sends the contents of the file to standard output which in this case goes through netcat and returns to the browser.
Voila! We should be able to navigate to the browser and navigate to 192.168.1.101:7999/index.html and see our simple html file. The only problem is that it will be rendered as raw text.
This is because when we send the output back to the browser, we send no headers and so the browser doesn't know what to do with the response and defaults to raw text.
In the next chapter we'll add the response headers and set up proper mimetype and content lengths!

Chapter 8 - To defeat the huns

We currently can handle requests and send back the correct files but because we aren't setting any headers, our responses are being displayed as raw text. Before we get to sorting this out, let's make our index.html page a little bit more complex.
./app/index.html

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <link rel="Shortcut Icon" type="image/x-icon" href="favicon.ico">
        <title>Fun!</title>
        <link rel="stylesheet" href="/css/style.css">
        <script src="/css/script.js"></script>
    </head>
    <body>
        <h1>Hello, World!</h1>

        <hr>
        <h2>Test PNG</h2>
        <img src="/images/test.png">
```

```
        <hr>
        <h2>Test JPG</h2>
        <img src="/images/test.jpg">

        <hr>
        <h2>Test MP4</h2>
        <video width="200" height="200" controls>
            <source src="/videos/test.mp4#t=0.01" type="video/mp4">
        </video>
    </body>
</html>
```

We have a relatively fully featured web page now with a number of requests. We have requests for a favicon, style.css, script.js, test.png, test.jpg, and test.mp4. All files are also located in their own resource specific file which are in turn all under the app directory.
Now we can see the problems we're going to be facing. We have all sorts of formats and extensions to deal with, we have image data and textual data, we also have a video to send over. If our server could handle all these types, it would be a capbale little server.
Let's get started!
The first thing we'll do is instead of calling cat in out handle function, we are going to create a new function called send_file.

```
...
handle_requested_resource() {

    regexp=".* (.*) HTTP"
    [[ "${request_headers[0]}" =~ $regexp ]]

    resource="${BASH_REMATCH[1]}"

    requested_resource="./app$resource"
    if [ -f "$requested_resource" ]
    then
        send_file "$requested_resource"
    fi
}
...
```

We are going to pass in the requested resource to our function and this send_file function will then take responsibility of setting up the headers and content length.

```
...
send_file() {
    # -> content_type
    requested_resource="$1"
    extension="${requested_resource##*.}"
    set_response_content_type "$extension"

    # -> data |  content_length
    get_requested_content "$1"

    # -> response_headers
    set_response_headers  "$content_type" "$content_length"
```

```
    # -> response
    build_response "$response_headers" "$content"

    # -> ECHO data | PRINTF data
    send_response "$response"
}
...
```

We first get the variable that was passed in, bash doesn't pass in named variables, each function can access the arguments through $1, $2, and so on depending on how many arguments are passed in.
We get the extension of the file using parameter expansion. Here the # means we are looking to match something. The * option will mean to remove. The extension command therefore means match up to including the first dot we see and remove those characters. The extension would be the result of this expansion.
This seems like a strange form of regex and it might be simpler or easier to understand if we use regex.

```
...
    regex=".*\.(.*)"
    [[ "$requested_resource" =~ $regex  ]]
    extension="${BASH_REMATCH[1]}"
...
```

The parameter expansion is more concise so let's use that one for now.

Setting the Content-Type Header

Now that we have the extension we can then set the Content-Type on the response we want to send. HTML files should be sent back as text/html and png should be image/png.

```
...
set_response_content_type() {
    case "$1" in
        "html")
            content_type="Content-Type: text/html"
            ;;
        "css")
            content_type="Content-Type: text/css"
            ;;
        "js")
            content_type="Content-Type: text/javascript"
            ;;
        "ico")
            content_type="Content-Type: image/x-icon"
            ;;
        "png")
            content_type="Content-Type: image/png"
            ;;
        "jpg" | "jpeg")
            content_type="Content-Type: image/jpeg"
            ;;
        "mp4")
            content_type="Content-Type: video/mp4"
            ;;
        *)
```

```
            content_type="Content-Type: text/plain"
            ;;
    esac
}
...
```

We pass in the extension and based on the extension we will set a content_type. We are using a case statement and bash uses a very strange looking case. Everything is unbalanced which is cool.

Getting the Content and setting the Content-Length Header

```
...
get_requested_content() {
    length=$(stat --printf "%s" "$1")
    content_length="Content-Length: $length"
    content=$(cat "$1" | sed 's/\\/\\\\/g' | sed 's/%/%%/g' | sed 's/\x00/\\x00/g')
}
...
```

We pass in our requested_resource and we will cat the file and escape certain things.We use the read the length before we escape the data as the escaping process is only so we can use printf later. We don't want to accidently count the escape characters as part of our Content-Length. We can use the stat command to get the length of the file and with that we are good to go.
To escape our data we use sed and replace some problematic characters. We will be using printf later and so we will need to escape slashes and percents as printf uses those characters. We also want to use sed to replace nulls with an escaped null. This way we can send the data along using printf. Sed works by going line by line doing a replace.
For images and videos, we will need to use printf to send the data, this is because of the characters used in them. Text files like html and CSS we can use straight echo.
Once we have our data, we then calculate the Content-Length using the bash builtin #. This will give us the length of the content that we are going to serve.
Now we have our Content-Type and Content-Length and our Content. We can now set the rest of our response headers.

Setting the Version, Date, and Connection Headers

```
...
set_response_headers() {
    version="HTTP/1.1 200 OK"
    date="Date: $(date)"
    connection="Connection: Closed"

    response_headers="${version}$CR_LF${date}$CR_LF${cookies}$CR_LF$1$CR_LF$2$CR_LF${connection}"
}
...
```

We create the http status line, we create the Date header and the connection header.
We then merge all the headers, including the headers we passed in, which was the Content-Type and Content-Length.

Adding the Headers and Content To form a Response

Now we have all of our headers set, we just need to add the headers and the content together.

```
...
build_response() {
    response="$1$CR_LF$CR_LF$2"
}
...
```

Voila! We have our response, we have the headers followed by 2 new lines and then we have the body which is the content.

Sending the Response

```
...
send_response() {
    printf -- "$1$CR_LF"
    exit
}
...
```

Here we using printf with no formatting to send the data and we pad the response with a final new line character as I was getting strange behavior when the exact size of the content was being sent back and the body contained just the file. I'm guessing there is something I don't understand with the way printf and escaping is working with how the data is being sent and what the browser expects.Padding seems to have resolved the issue and unified the logical different I had between regular files and images.
Once we send the data back we will exit the script and return to sitting and listening for another connection.
We should now be able to refresh our index.html page and see a fully featured page with all of our html loaded, our images loaded, our css loaded and our javascript loaded.
! We have now put together a very barebones web server!
Now that we have files being requested and handled properly, let's look at sending POST requests to our webserver.

Chapter 9 - Going Postal

Currently we can handle GET requests but we don't have any way of dealing with POST requests. Let's fix that.
Let's first write up a basic login page.

```
...
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <link rel="Shortcut Icon" type="image/x-icon" href="favicon.ico">
        <title>Manga Readers Club - Login!</title>
        <link rel="stylesheet" href="/css/style.css">
    </head>
    <body>
        <h1>Manga Readers Club - Login!</h1>

        <hr>

        <form action="/login" method="POST">
            <label>Username</label>
            <input type="text" name="username">

            <br>
            <label>Password</label>
            <input type="password" name="password">

            <br>
            <input type="submit" value="Submit">
        </form>
    </body>
</html>
...
```

Here we have a form that takes in a username and password and submits it as a POST request to /login.
Now we need to parse our the body of a request if it is a POST request.

```
...
process() {
    IFS=$LF

    get_request_headers

    get_request_body

    handle_requested_resource
}
...
```

Now we add a step where after we get the headers, we will get the body.

```
...
get_request_body() {
    request_type="$(echo "${request_headers[0]}" | cut -d" " -f1)"

    post_length=0
    for i in "${request_headers[@]}"
    do
        header=$(cut -d":" -f1 <<< "$i")
        if [ "$header" = "Content-Length" ]
        then
            post_length=$(echo "$i" | cut -d":" -f2 | tr -d "$CR" | tr -d "$LF" | tr -d ' ')
        fi
    done

    if [ "$post_length" -ne 0 ]
    then
        IFS= read -n "$post_length" request_body
    fi
}
...
```

When this function gets called, we have an array of headers and so we can check if the first header is a post request. Here we are using the cut option to parse out the type of the request but we could also use regex here.Next we need to loop through the headers to find the Content-Length header. Like our response, POST requests will have a content length that we can use to figure out how much data is in the body.
We loop through looking for the Content-Length header and once we find it, we parse the length out. Here we do some cutting and trimming where we remove all newline characters.
Once we have the length we then read more data in from the standard input. This is because we processed everything in the standard input until we hit a blank new line. Once we hit this line we returned back to our script and so if there is a body it would stay in the standard input.
We set the IFS here to nothing as the body of a http response could contain newlines and we want to process everything in the body. We also use the -n flag so that we specifiy how many bytes we want to grab from standard input.

```
handle_requested_resource() {
```

```
    regexp=".* (.*) HTTP"
    [[ "${request_headers[0]}" =~ $regexp ]]

    resource="${BASH_REMATCH[1]}"

    requested_resource="./app$resource"
    if [ -f "$requested_resource" ]
    then
        send_file "$requested_resource"
    fi

    log "$request_body"
}
```

Here we add a line after our if statement where we will output the request body as it was parsed out.
Voila! We have wired up POST requests, now let's set up route end points so that we can do something useful with our POST request.

Chapter 10 - Who doesn't need a Dictionary

Now that we have post requests going let's set up routes. This way we can make requests to /login instead of /login.html. Doesn't that look better already!
The function dictionary is really a dispatcher. It looks at the url and based off the url it dispatches to the correct function.

```
...
declare -r DEBUG=1

declare -A function_dictionary=(
    [login]=login
)
...
```

The first thing we're going to do is a set up a function dictionary hashmap in bash. Luckily the storing of functions is easy in bash as functions don't really take arguments and we can call function by just using its name.Here we are saying if we get a route of login, then call login.
In our request handling, we currently check to see if the requested resource exists. Now if the resource doesn't exist, we'll check the function dictionary.

```
...
handle_requested_resource() {
    regexp=".* (.*) HTTP"
    [[ "${request_headers[0]}" =~ $regexp ]]

    resource="${BASH_REMATCH[1]}"

    requested_resource="./app$resource"
    if [ -f "$requested_resource" ]
    then
        send_file "$requested_resource"
    fi

    requested_resource="${resource:1}"

    for x in "${!function_dictionary[@]}"
    do
        if [[ "$requested_resource" =~ $x ]]
```

```
        then
            ${function_dictionary[$x]}
        fi
    done

    send_file "./app/404.html"
}
...
```

Here we are going to loop through the function dictionary and check to see if we have any matches in the dictionary for the route we are trying to access. We use the =~ operator to do a regex match. We'll need to do this so we can handle dynamic routes later.When we make a request we do so by gonig to /login, so we need to chop off the first character which is what we do with the colon option.
This is a hashmap so we don't need to use a loop to check the keys but we'll need the loop later when we want to deal with variable routes like /user/1 and /user/2.
Now let's write our login function!

```
...
login() {
    if [ "$request_type" = "GET" ]
    then
        send_file "./app/login.html"

    else
        username=$(echo -n "$request_body" | cut -d'&' -f1 | cut -d'=' -f2)
        password=$(echo -n "$request_body" | cut -d'&' -f2 | cut -d'=' -f2)

        if [ "$password" = "123" ]
        then
            session_id=$(uuidgen)
            touch "./sessions/$session_id"
            cookies="Set-cookie: session_id=$session_id"
        fi
        send_file "./app/account.html"
    fi
}
...
```

Now we can check if our request is a GET or a POST. The GET is simple, we simply want to return the login.html page we already wrote.If the request is a post however, we can parse out the login information and log the person in!
We need to create a sessions folder next to our app and once a user is logged in we create a session token for them. We are also setting a cookie but that's not being used yet.

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <link rel="Shortcut Icon" type="image/x-icon" href="favicon.ico">
        <title>Manga Readers Club - Account!</title>
        <link rel="stylesheet" href="/css/style.css">
    </head>
    <body>
        <h1>Hello</h1>
```

```
        <hr>
        <div><a href="/comics">Comics</a></div>
    </body>
</html>
```

We should be able to navigate to /login and try logging with any username and with the password 123 and we should get the account.html page printed!
With that we have our end points starting to work! Now in the next chapter let's wire up templating. This way we can have bash commands in our html files and do all sorts of cool things!

Chapter 11 - Bashruptcy

In our account page, let's add a simple username display so we can see who we logged in as.
./app/account.html

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <link rel="Shortcut Icon" type="image/x-icon" href="favicon.ico">
        <title>Manga Readers Club - Account!</title>
        <link rel="stylesheet" href="/css/style.css">
    </head>
    <body>
        <h1>Hello, "$username"</h1>
        <hr>
        <div><a href="/comics">Comics</a></div>
    </body>
</html>
```

Here we have a raw bash variable embedded in out html. We are going to use bash to consider this as a regular bash string and it will then run all the bash code within it.This way we don't have to create our own templating language! We can just reuse bash!
Our strategy is gonig to be that we request a page that is a template. So we need to get the contents of that file, run all the code in it and generate some html and then we send that html back out.

```
...
render_template() {
    template=$(eval "cat <<- END
    $(cat "$1")
END
")
}
...
```

The first step is our render_template function. Here we are gonig to call eval against the contents of the file we are trying to use as the template. We want to handle multiline input and so we use the redirect where we specifiy the end of file marker explicitly.

```
...
send_html() {
    content="$1"
    content_length="${#content}"
```

```
    set_response_content_type "html"
    set_response_headers "$content_type" "$content_length"
    build_response "$response_headers" "$content"

    send_response "$response"
}
...
```

The second step is to write a send_html. We have a send_file function but that sends a file with no transformations, instead we want to send straight html that we generate and so we can use this function for that.

```
...
login() {
    if [ "$request_type" = "GET" ]
    then
        content="$(cat ./app/login.html)"
        send_html "$content"

    else
        username=$(echo -n "$request_body" | cut -d'&' -f1 | cut -d'=' -f2)
        password=$(echo -n "$request_body" | cut -d'&' -f2 | cut -d'=' -f2)

        if [ "$password" = "123" ]
        then
            session_id=$(uuidgen)
            touch "./sessions/$session_id"
            cookies="Set-cookie: session_id=$session_id"
        fi
        render_template "./app/account.html"
        send_html "$template"
    fi
}
...
```

Now we can replace our send_file commands with a call to render_template and send_html.
Voila! We should now be able to go to the browser and login again and this time be greeted with "Hello, Username"!
The eval is terrifying as we can do all sorts of things inside our html. Very cool.
Now that we have the function dictionary and the templating done, we have the major pieces of a web server working. The last major piece we'll work on is session management. Currently we set up the session but do nothing with it. Let's fix that!

Chapter 12 - Rememberings who we are

Currently when a user logs in we create a session token and use the file system to store the session. This is a quick and dirty way to get everything going.

```
...
login() {
    if [ "$request_type" = "GET" ]
    then
        content="$(cat ./app/login.html)"
        send_html "$content"

    else
```

```
        username=$(echo -n "$request_body" | cut -d'&' -f1 | cut -d'=' -f2)
        password=$(echo -n "$request_body" | cut -d'&' -f2 | cut -d'=' -f2)

        if [ "$password" = "123" ]
        then
            session_id=$(uuidgen)
            touch "./sessions/$session_id"
            cookies="Set-cookie: session_id=$session_id"
        fi
        render_template "./app/account.html"
        send_html "$template"
    fi
}
...
```

This is just to go over what we've done. Now we need to wire up the cookies to our responses so that it gets sent back to the browser. After that we also need parse the cookie specifically when we read the request headers.

```
...
set_response_headers() {
    version="HTTP/1.1 200 OK"
    date="Date: $(date)"
    connection="Connection: Closed"

    if [ "$cookies" = "" ]
    then
        response_headers="${version}$CR_LF${date}$CR_LF$1$CR_LF$2$CR_LF${connection}"
    else
        response_headers="${version}$CR_LF${date}$CR_LF${cookies}$CR_LF$1$CR_LF$2$CR_LF${connection}"
    fi
}
...
```

The first thing we do is set the cookie header if there are cookies to send back.Now that we are setting the cookie header in the response, we can look at parsing the request headers for the session token.
This function should ultimately be merged with the get_request_headers as the logic is duplicated. We really should parse as we read. I fell into the temporal decomposition pitfall, where because reading and parsing are two steps for me, I split them out. In this case they would make sense to keep together as reading the headers really does require parsing them as there are headers we need to get the body. This would also result in query parameters from get requests and parameters from post requests being handled in the same function and could be put into the same variable in the future.

```
...
get_request_body_cookies() {
    request_type="$(echo "${request_headers[0]}" | cut -d" " -f1)"

    post_length=0
    for i in "${request_headers[@]}"
    do
        header=$(cut -d":" -f1 <<< "$i")
        if [ "$header" = "Content-Length" ]
        then
```

```
            post_length=$(echo "$i" | cut -d":" -f2 | tr -d "$CR" | tr -d "$LF" | tr -d ' ')

        elif [ "$header" = "Cookie" ]
        then
            regex=".*session_id=(.*);?"
            [[ "$i" =~ $regex ]]
            session_id=$(echo "${BASH_REMATCH[1]}" |  tr -d "$CR" | tr -d "$LF")
        fi
    done

    if [ "$post_length" -ne 0 ]
    then
        IFS= read -n "$post_length" request_body
    fi
}
...
```

Here we are renaming our get_request_body to get_request_body_cookies as this function will also handle our cookies now. We originally used this function to get the Content-Length so it makes sense to use this function for our cookies as well.
We now check to see if we get a Cookie header and if we do we can then use regex and capture groups to get the session_id from the cookie the request sent back.
Now that we have the id, we can create a function that will check for a session.

```
...
check_session() {
    if [ ! -f "./sessions/$session_id" ]
    then
        render_template "./app/login.html"
        send_html "$template"
    fi
}
...
```

This is a helper function we can call at the beginning of a route when we want to make sure a user is logged in before they access the function. If they aren't logged in, we will ask them to log in.Before we test our session handling, let's add a logout link and set up the route to log out properly.

```
...
declare -A function_dictionary=(
    [login]=login
    [signout]=signout
    [account]=account
)
...
```

Here we add the signout and account function to our function dictionary so we can match against it.

```
...
signout() {
    if [ -f "./sessions/$session_id" ]
    then
        rm "./sessions/$session_id"
        session_id=""
```

```
        cookies="Set-cookie: session_id=$session_id"
    fi
    render_template "./app/login.html"
    send_html "$template"
}
account() {
    check_session
    render_template "./app/account.html"
    send_html "$template"
}
...
```

The logout function is quite straightforward, we simply check to see if the session exists in the session folder and if it does we delete the session, reset the cookie and then redirect back to the login page.Our account function calls check_session before returning the account page.
Now we just need to add the signout link to our account page.

```
...
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <link rel="Shortcut Icon" type="image/x-icon" href="favicon.ico">
        <title>Manga Readers Club - Account!</title>
        <link rel="stylesheet" href="/css/style.css">
    </head>
    <body>
        <h1>Hello, "$username"</h1>
        <hr>
        <div><a href="/signout">signout</a></div>
        <div><a href="/comics">Comics</a></div>
    </body>
</html>
...
```

Voila! We should now be able to navigate to /account and be shown the login screen. We should be able to login and see our account page properly and finally we should be able to click sign out and be logged out. We can then navigate to /account again to verify that we indeed can't get through.
With that we have a major piece of our web server up and working!

Chapter 13 - Here we are

Now that the core logic of our server is working, let's write a simple little comic site that will list all the comics available that we can click into to get a chapter listing which if we click into once more we can read that chapter. For this we're going to use more templating logic and also add in rewrite rules so that we can do shorthands like /comic/comic_name/1 to mean chapter 1 of comic_name.
The first thing we'll do is set up a folder in our app directory called comics. Inside comics we have multiple series that we want to make available. Inside the series folder we will have folders for each chapter. Inside each chapter is a set of images that is the comic book. We're going to assume everything is named properly such that the default sort will get us the order we want for all the listings.
We could make this more robust by using a database but this is just a simple application to show that our bash server is useful!

The Comics Page!

Currently our account page links to /comics but it doesn't go anywhere, let's fix that.

```
declare -A function_dictionary=(
    [login]=login
    [signout]=signout
    ["comics"]=comics
)
```

We add comics to our function dictionary so that we can get to the comics endpoint.

```
comics() {
    check_session
    render_template "./app/comics.html"
    send_html "$template"
}
```

The comics end point will render the comics template after checking the session. The session checking isn't really needed here but let's use it because we have it.

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <link rel="Shortcut Icon" type="image/x-icon" href="/favicon.ico">
        <title>Directory - Comic Book Readers Club!</title>
        <link rel="stylesheet" href="/css/style.css">
    </head>
    <body>
        <h1>Directory - Comic Book Readers Club!</h1>
        <div>
            <a href="/signout">signout</a>
        </div>
        <hr>
        $(
            cd ./app/comics
            for id in *
            do
            echo "<div><a href='/comics/$id'>$id</a></div>"
            done
        )
    </body>
</html>
```

Here we have html and bash being mixed together to generate html. We are looping through all the files in app/comics and creating a link for each one.

The Issues Endpoint

Now that we have a list of comics, let's set up the specific chapter listings for a comic. Now we need to have some dynamicness so we can handle different comics with just one function.

```
declare -A function_dictionary=(
    [login]=login
    [signout]=signout
    ["^comics$"]=comics
    ["^comics/(.*)"]=issues
)
```

Here we have our first regex match in our function dictionary. We also have a capture group. This route will handle anything in the format of comics/SOMETHING.Let's see how this works in action!

```
issues() {
    check_session
    comic_name="${BASH_REMATCH[1]}"
    render_template "./app/issues.html"
    send_html "$template"
}
```

Here we parse out the comic name using BASH_REMATCH at position 1. This is because when we loop through the function dictionary in our request handler, we do =~ which does a regex test. If the test succeeds we run the function. If we have capture groups, we also get that in BASH_REMATCH. This makes it very easy to deal with dynamic routes.

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <link rel="Shortcut Icon" type="image/x-icon" href="/favicon.ico">
        <title>$comic_name Chapters!</title>
        <link rel="stylesheet" href="/css/style.css">
    </head>
    <body>
        <h1>$comic_name Chapters!</h1>
        <hr>
        $(
            cd ./app/comics/$comic_name
            counter=0
            for chapter in *
            do
                counter=$((counter+1))
                echo "<div><a href='/comics/$comic_name/$counter'>$chapter</a></div>"
            done
        )
    </body>
</html>
```

Our template also has access to variables we set in our issues function so we can cd into the correct comic we want to read and get a list of all the chapters within that folder. Voila! We now have chapter listings.

The Issue Page

Finally we can get to the last screen in our comic bookn reader, that is the issue page. This page will show the images of the chapter we want to read. Here we need to parse 2 variables from the link from the issues page. We need the comic name and the chapter number.

```
...
declare -A function_dictionary=(
    [login]=login
    [signout]=signout
    ["^comics$"]=comics
    ["^comics/(.*)"]=issues
    ["^comics/(.*)/(.*)"]=issue
```

```
)
...
```

We add one final route for our issue page and we set the regex to have 2 capture groups as we want to capture the comic name and the chapter number.

```
...
issue() {
    check_session
    comic_name="${BASH_REMATCH[1]}"
    issue_number="${BASH_REMATCH[2]}"
    render_template "./app/issue.html"
    send_html "$template"
}
...
```

We can access the comic name via the first BASH_REMATCH and the second has the issue number. With these 2 pieces of data we can render the final page.

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8">
        <link rel="Shortcut Icon" type="image/x-icon" href="/favicon.ico">
        <title>$comic_name - Issue $issue_number</title>
        <link rel="stylesheet" href="/css/style.css">
        <script type="text/javascript" src="/js/issue.js"></script>
    </head>

    <body>
        <h1>$comic_name - Issue $issue_number</h1>
        <hr>
        $(
            files=(./app/comics/"$comic_name"/*)
            folder=("${files["$issue_number-1"]}"/*)

            if [ -d ${folder} ]
            then
                folder=("${folder}"/*)
            fi

            counter=0
            for page in ${folder[@]}
            do
                counter=$((counter+1))
                page=$(echo "$page" | sed 's/\.\/app//g')
                if [[ "$counter" -le 3 ]]
                then
                    echo "<div><img src='$page' height=1400></div>"
                else
                    echo "<div><img realsrc='$page' height=1400></div>"
                fi
            done
```

```
        )
    </body>
</html>
```

This page is a little bit more involved as we are loading images and we need to deal with paths correctly to get everything working.Voila! We should now be able to navigate /login and go through the entire login and go to the comics and begin reading. We now have a little web server written entire in bash!
