CY350 - Computer Networks Project #2 –

Web Server Using http.server

In this programming assignment you will build upon class materials and your work in project 1 to create a small but functional web server that accepts connections from a client and responds appropriately to GET requests. Your server will need to respond to the client with the correct HTTP headers for each request it receives. There are a variety of cases you will need to handle, but by breaking it into a series of subproblems you should find that it is very doable!

Project Due Dates:

Part 1 – 2359 06 MAR 24

Part 2 – 2359 17 MAR 24

Limitations:

Code: You must write your code in Python3 and you must use the **http.server** module. You may use any other non-network modules you need. Please consult your instructor if you are unsure if a module is allowed.

Autograder: This assignment will be graded via a Gradescope autograder, and as a result there are some requirements you must meet. You must name your file in accordance with the file name requirements. You must follow the specifications for the 5 required functions below as well. **Note:** We will manually review any submission that does not get a perfect score to assign partial credit as merited.

Hidden Tests (New): With moving into part 2 of this project not every test case will be visible to you. This is by design. Do not send the instructional team messages telling us that the autograder is not working because you can't see the test. These tests are just to ensure that you are not hard coding solutions into your code.

Printing (Changed): In our continual efforts to make your experience in our course better, the autograder has been updated. Congratulations, you can now print out as much as you want without breaking anything! You can also continue to use the provided **self.debug** function if you want to maintain the ability to turn off all of your messages at once.

Submission Limitations: You may submit an unlimited number of times. Your best score will be kept.

Names: You must name your file **project2_server.py** . Failure to do so will result in failing the autograder. We could fix this, but honestly it is on you to **actually read and follow** the instructions. You must leave the name of your class **SimpleServer**. Failure to do so will result in

failing the autograder. Do not change the names of any of the required functions. Failure to do so will result both in failing the autograder and also your code breaking because the underlying Class expects those function names. If you choose to do the second bonus portion of the assignment you must name the file **unauthorized.html.**

General Information:

1) Your server will use port "8080".

2) HTTP headers you will implement:
    - **Response Code** (i.e., 200, 304,…)
    - **Content-Length** (this is NOT the length but actually the size)
    - **Content-Type** (see below but look up http.server documentation for specifics)
    - **Last-Modified** (appropriate date/time format)
    - **Connection** (Open or Closed)
    - **Location** special case (301): (new location and/or name)

3) Content Type you will handle:
    - Text documents: "text/html", "text/css", "text/csv", "text/xml"
    - Images: "image/jpg", "image/png", "image/x-icon"
    - Videos: "video/mp4"

4) You will be using the http.server module to abstract away the issues of handling TCP connections and leaving you to instead focus on building the logic for handling and responding to GET requests. Specifically, you will be creating a subclass of http.server.BaseHTTPRequestHandler which will require you to build the logic for several functions (see Part 1 and the provided code file for more information) that will actually manage requests and responses. While you don't need a deep understanding of everything the http.server module is doing for you, it is still important to review how HTTP requests work so that you can work through the logic.

5) We have kept the information for Part 1 in here so that you can reference it as you build out the additional functionality for Part 2.

Part 1 – Handling Basic GET Requests

Before you worry about handling the various corner cases and specific situations it is probably important to get a webserver up and running and serving basic content and responses. This will be your focus for Part 1, and we have broken it into some subproblems for you to tackle. First, let's get oriented to the functions you will be implementing (see the code file for additional information).

Necessary Functions:

- debug(self, message):
    - The debug function has been provided to you and you should use it anywhere that you want to print something out for your own testing and development.

- You can enable the debug function's printing messages by setting debug_state = True inside the debug function. You can disable the printing of messages by setting debut_state = False.
- Instead of using print("your message"), simply use the command self.debug("your message").
- Ensure you set debug_state = False before submitting or you will fail the majority of tests.
- **NEW:** Even though you don't need to use it for part 2, **DO NOT** delete the debug function. The autograder is still looking to make sure it is there (definitely not because we were lazy and forgot to remove that part of the code) so you need to make sure it is still there!

- determine_response_code(self):
  - This function will be used to determine what response code should be generated when you receive a GET message.
  - In part 1 you only need to worry about 200 and 404 messages. This will be extended in part 2.
  - Do not add any extra arguments to the function beyond "self".
  - See the provided code file for additional information.

- determine_content_type(self):
  - This function contains the logic to determine what type of file is being sent back to the client. Think through how we can generally determine the type of file by looking at it.
  - Do not add any extra arguments to the function beyond "self".
  - See the provided code file for additional information.

- send_response_headers(self):
  - This function contains the logic to build the appropriate header messages depending on how you have handled the messages in other functions. It is worth noting that you don't need to implement the actual message being sent to the client, the underlying http.server module will handle that. Instead, you are writing these messages to the appropriate buffer which will all be sent when triggered.
  - Do not add any extra arguments to the function beyond "self".
  - See the provided code file for additional information.

- do_get(self):
  - This function serves as the control function that is triggered automatically by the http.server class upon receipt of a GET message. It should be used to call the other functions as appropriate in order to build the response.
  - Do not add any extra arguments to the function beyond "self".
  - See the provided code file for additional information.

Additional Functions:

You may build any additional helper functions you believe are necessary or useful to you so long as they are contained within the SimpleServer class. Any function that is outside of the SimpleServer class you are building will not be imported into the autograder and as a result you will lose points. If you are uncertain about what this means, please talk to an instructor.

Part 1a: Serve the Root Page

Now that you have some idea on the functions involved, it is time to break the big problem of making a web server into smaller subproblems that you can tackle. Here is our suggestion on how you should approach this, but you are free to tackle this however you choose:

- Problem 1: Get the server up and connecting so that you can see GET messages.
  - In order to solve this problem you will need to leverage the existing skeleton code as well as do some research into how messages are handled by the http.server. BaseHTTPRequestHandler class.
  - You should be able to request a resource (either from a browser or the command line) and then use self.debug to print out the relevant message.
- Problem 2: Use the information contained in the GET request to determine the correct response code for files at the root directory (the same level as your server code)
  - A request for "/" should be translated as asking for index.html.
  - At this point you are only handling 200 OK and 404 Not Found responses.
- Problem 3: Use the information from the GET request and your solution to Problem 2 to determine what the content type for the requested resources is.
- Problem 4: Use the information you have created (and that already exists in the class) to generate the headers.
- Problem 5: Make sure that do_GET correctly calls the other functions in a way that generates a valid response.
  - You will know you have solved all of these when you can send a request to your server's IP address and it successfully results in displaying the index.html page we have provided and if you send a request for a resource that doesn't exist and it displays the notfound.html page. Note: The connection should be closed once you have sent back the resource. You will fail multiple tests if you do not close the connection.

Part 1b: Serve Other Resources

Now that you can correctly serve a request for index.html (or for notfound.html if appropriate) you need to make sure that your server can return all the other content. You don't need to worry about handling requests for things that have moved yet, for now you can assume that you either get a valid path to a resource or it is an invalid request.

- Problem 6: So far everything you have done is at the root level of the server (at the same level as your code), but there are some files located in subdirectories. You still need to be able to generate all of the correct information for resources at different levels.
- Problem 7: test2.html requires multiple requests and various types of content to be handled to properly load the website in a browser. Make sure that your functions are dynamically handling the generation of the response code, content type, and headers. Don't hard code any of these!

Part 1 – Hints

- **Start early.**
- Take advantage of the autograder and its ability to give you feedback.
- **Start early!**
- Before you start to code anything, make sure you understand what each function is expected to do and build a plan.
- In addition to the autograder you should be running tests of your own at every step. Is the function generating the outputs you expect? Have you checked what type each value is?
- **START EARLY!!!!**
- Read the comments in the starter code carefully. There are fewer of them than in project 1, but they are there to help you.
- You can test your code by starting the server on your machine and using a browser to navigate to 127.0.0.1 . If the server is working correctly it should display index.html (or whatever other resource you have requested).

Part 1 – Files

The zip file for your project contains the following files:

- project2_server_starter.py - This is the only file you need to edit / submit.
- index.html
- test1.html
- test2.html
- moved - You won't need this until part 2
- redirect.html
- favicon.ico
- notfound.html
- pages/
    - test3.html
- images/
    - heavy_drop.jpg
    - JNN_STT_airfield.jpg
    - test.mp4

Part 1 – Submission

You will submit your code file (and only your code file) to the Project 2 Part 1 assignment located in Gradescope. You will find the link to Gradescope inside our Canvas course page. As stated above, the file **must** be named **project2_server.py** . Once you have submitted successfully you will receive an automatically generated email confirming your submission and you will be able to see the results of the autograder (it shouldn't take more than a minute). We have chosen to enable visibility for all tests and for the stdout so that you can use it for troubleshooting.

Once you are comfortable with your grade in Gradescope (i.e., your final submission), you will also need to go to the corresponding assignment in Canvas and fill out the academic statement. As per usual, any assistance you receive should be noted with in line citations.

Part 2 – Expanding Functionality (The New Stuff)

Now that you have implemented a basic webserver in part 1, we need to add some additional functionality. In part 2 of this project, you will be adding the ability to handle conditional GET messages as well as logic to handle files that may have been moved. You will also have the opportunity to implement some basic security as a way to earn a few bonus points. Before we dive into the specific aspects of part 2, a quick recap on some part 1 information:

Necessary Functions:

All functions from part 1 are still necessary, **to include the debug function** even if you choose not to use it in part 2. Please feel free to refresh yourself on their purposes in the documentation above before proceeding.

Additional Functions (no change):

You may build any additional helper functions you believe are necessary or useful to you so long as they are contained within the SimpleServer class. Any function that is outside of the SimpleServer class you are building will not be imported into the autograder and as a result you will lose points. If you are uncertain about what this means, please talk to an instructor.

Part 2a: Conditional Get Messages

At this point you should already be serving pages for GET requests (that was what part 1 was), however it is a bit wasteful of resources to continuously send resources to a client if nothing has changed since the last time they requested that resource. You are now going to implement some logic that will determine if your server should send a resource or if it should simply inform the client that nothing has changed.

- Problem 1: Determine if a message includes a conditional GET.

    - Remember that a GET request can also include some headers. You should probably figure out which one you need to look at. A good place to reference would be the HTTP lesson slides.
- Problem 2: Update your logic to reflect accordingly.
    - If there is no conditional GET, you should probably just proceed as though it is a new request.
    - If there is a conditional GET you need to update your logic to determine if the file has been updated since the time provided in the request.
        1. If yes, send the file.
        2. If no, what do you need to update and send back instead? (Hint: This would be a new status code. Take a look at the general information.)
- Problem 3: Since we are now handling conditional GET messages you need to account for a new header! We should proactively provide information about the last time a file was modified when we send a resource.
    - Create the **'Last-Modified'** header and include it in the list of headers generated in the **send_response_headers** function.

Part 2b: Files That Have Moved

In addition to conditional GET messages, we also need to account for the reality that as we update our server sometimes files will be moved or renamed or even removed entirely. We can't expect our users to keep track of that, so instead we have the **moved** file which is a simple system. Each row of the file represents a resource, and the first value is the original name / location, and the second value is the updated name / location.

- Problem 4: Determine if a requested resource has been moved.
    - You should already be checking if a requested resource is valid. In part 1 either a resource existed (200) or it didn't (404). Now you need to extend that logic further: if a resource doesn't exist you now need to see if it has been moved.
- Problem 5: Once you have determined if a resource has been moved or not, you need to update your logic to handle it accordingly.
    - If it has not been moved and does not exist treat it like a 404 error and handle it accordingly.
    - If it has been moved, you need to assign a different response code (see general information above) and update your logic to provide the new location.
        1. This will require you to create a new header: **'Location'**
        2. This header should be created only if a file has been moved!
        3. You do not need to send the file if it has been moved, instead you will provide the location to the client and they will send a new request for the updated resource.
    - Do not hard code this! The autograder has a different **moved** file than you do as well as additional resources to check. Your code must update and check dynamically.

Part 2c: Let There Be Bonus Points!

This is a cyber class, and as part of any cyber class we should be at least thinking about security. Right now your server is configured to allow any user to request any resource and the only limitation is whether or not a resource that exists. We should probably implement something to restrict access to some of those files.

- Problem 6: Implement some very weak security by preventing a request for the file **moved** from returning the file and instead generating a **401** error. A 401 error represents "Unauthorized".
  - This is a bonus problem so that's about all the hints you are getting.
  - Properly implementing this will net you 5 bonus points.
- Problem 7: While we could just return a 401 error by itself, that isn't very fun. Implement a custom HTML response object (which is a fancy way of saying an HTML page that is only used for this response) that will be sent back to the client if they request an unauthorized resource.
  - It must include the words **401 – UNAUTHORIZED** somewhere in the page.
  - You can see an example in our **notfound.html** page.
  - You get 1 bonus point if you just make a boring generic page like the provided **notfound.html** page, but you can earn up to 5 bonus points if you make it fancy.

Part 2 – Hints

- **Actually start early this time.**
- Take advantage of the autograder and its ability to give you feedback.
- **No, seriously, start early this time!**
- You should now understand what the existing functions do, so before you start to code anything take a moment and plan through which functions need to be updated to add in the new functionalities. You will find that most of them require modifications to more than one function.
- **WHY HAVEN'T YOU STARTED?**
- In addition to the autograder you should be running tests of your own at every step. Is the function generating the outputs you expect? Have you checked what type each value is? Wireshark is your friend here.
- **We don't know how else to put it: START EARLY!!!!**
- You will probably have more luck asking your instructors for help than telling Jodel you don't know what to do.

Part 2 – Submission

- **Option 1 (No bonus):** You will submit your code file (and only your code file) to the Project 2 Part 2 assignment located in Gradescope. You will find the link to Gradescope inside our Canvas course page. As stated above, the file **must** be named **project2_server.py** . Once you have submitted successfully you will receive an

automatically generated email confirming your submission and you will be able to see the results of the autograder (it shouldn't take more than a minute). We have chosen to enable visibility for most tests and for the stdout so that you can use it for troubleshooting.

- **Option 2 (Bonus):** You will submit your code file **AND** your created webpage to the Project 2 Part 2 assignment located in Gradescope. **DO NOT ZIP THESE FILES.** You will find the link to Gradescope inside our Canvas course page. As stated above, the python file **must** be named **project2_server.py** and the HTML response object must be named **unauthorized.html**. Once you have submitted successfully you will receive an automatically generated email confirming your submission and you will be able to see the results of the autograder (it shouldn't take more than a minute). We have chosen to enable visibility for most tests and for the stdout so that you can use it for troubleshooting.

- **Both Options:** Once you are comfortable with your grade in Gradescope (i.e., your final submission), you will also need to go to the corresponding assignment in Canvas and fill out the academic statement. As per usual, any assistance you receive should be noted with in line citations.

Part 2 – Files

The zip file for your project contains the following files:

- project2_server.py - Provided after part 1 submissions.
- index.html
- test1.html
- test2.html
- moved - This is part 2 and you now need this.
- redirect.html
- favicon.ico
- notfound.html
- pages/
    - test3.html
- images/
    - heavy_drop.jpg
    - JNN_STT_airfield.jpg
    - test.mp4