# 

[TritonHTTP Specification](#_30j0zll)

[Project overview](#_1fob9te)

[Basic web server functionality](#_3znysh7)

[Project details](#_2et92p0)

[Mapping relative URLs to absolute file paths](#_tyjcwt)

[Supporting MIME types](#_3dy6vkm)

[Setting up virtual hosting](#_qiu14tum8327)

[Executable](#_1t3h5sf)

[Experimentation](#_4d34og8)

[Testing strategies](#_2s8eyo1)

[Grading](#_17dp8vu)

[Correctness/functionality: 75%](#_3rdcrjn)

[Testing strategy and completeness: 25%](#_26in1rg)

[Submitting your work](#_35nkun2)

[Submission guidelines:](#_1ksv4uv)

[Due date/time](#_44sinio)

[Starter code](#_2jxsxqh)

Project 2

In this project, you are going to build a simple web server that implements a subset of the HTTP/1.1 protocol specification called TritonHTTP.

# TritonHTTP Specification

The TritonHTTP spec is located in a [separate document](https://docs.google.com/document/d/1CcwzZ8LTNcBEpO3XFBmespEztZk_QHTBE19DaQYxGHA/edit?usp=sharing).

# Project overview

## Basic web server functionality

At a high level, a web server listens for connections on a socket (bound to a specific port on a host machine). Clients connect to this socket and use the TritonHTTP protocol to retrieve files from the server. Your server will read data from the client, using the framing and parsing techniques discussed in class to interpret one or more requests (if the client is using pipelined requests). Every time your server reads in a full request, you will service that request and send back a response back to the client. After sending back one (or more) responses, your server will either close the connection (if instructed to do so by the client via the “Connection: close” header, described below, or after an appropriate timeout occurs (also described below). Your web server will then continue waiting for future client connections. Your client should be implemented in a concurrent manner, so that it can process multiple client requests overlapping in time.

# Project details

Below are some of the details on implementing your project.

## Mapping relative URLs to absolute file paths

Clients make requests to files using a Uniform Resource Locator, such as /images/cyrpto/enigma.jpg. One of the key things to keep in mind in building your web server is that the server must translate that relative URL into an absolute filename on the local filesystem. For example, you might decide to keep all the files for your server in ~aturing/cse101/server/www-files/, which we call the document root. When your server gets a request for the above-mentioned enigma.jpg file, it will prepend the document root to the specified file to get an absolute file name of ~aturing/cse101/server/www-files/images/crypto/enigma.jpg. You need to ensure that malformed or malicious URLs cannot “escape” your document root to access other files. For example, if a client submits the URL /images/../../../.ssh/id\_dsa, they should not be able to download the ~aturing/.ssh/id\_dsa file. If a client uses one or more .. directories in such a way that the server would “escape” the document root, you should return a 404 Not Found error back to the client. However, it is valid if the client requests a URL like ~aturing/cse101/server/www-files/../../../cse101/server/www-files/images/crypto/enigma.jpg, where the URL does not escape from the document root.

Golang has a function called path.Clean() that will take a file path and clean up duplicate slashes, ‘..’s, etc. You will likely find it useful for this aspect of the project:

| fmt.Println(path.Clean("/subdir1/../subdir2/index.html")); |
| --- |

Results in:

/subdir2/index.html

## Supporting MIME types

Most applications interpret the contents of files based on their extensions (e.g. homework.txt represents an ASCII text file, whereas icon.jpg represents a JPEG image). Historically, file extensions have not played much of a role with the HTTP protocol, in part because sometimes web servers will dynamically generate content on demand. So, for example, a request line of the form

| GET /cgi-bin/genhaiku.pl HTTP/1.1 .... |
| --- |

Might invoke an external helper program (in this case, written in Perl), to generate a haiku poem on demand. But how to interpret the data that is returned? Is it ASCII text? A Word file? An image with the poem written in the center?

Web servers explicitly indicate the type of file via the “Content-Type” header. Examples include:

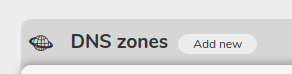
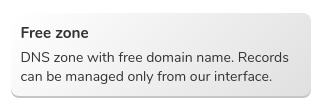
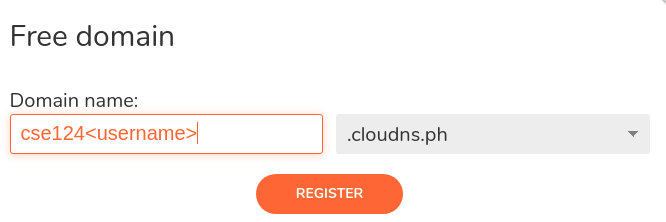
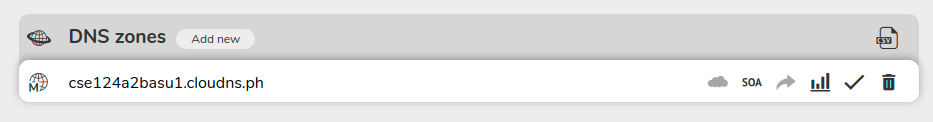
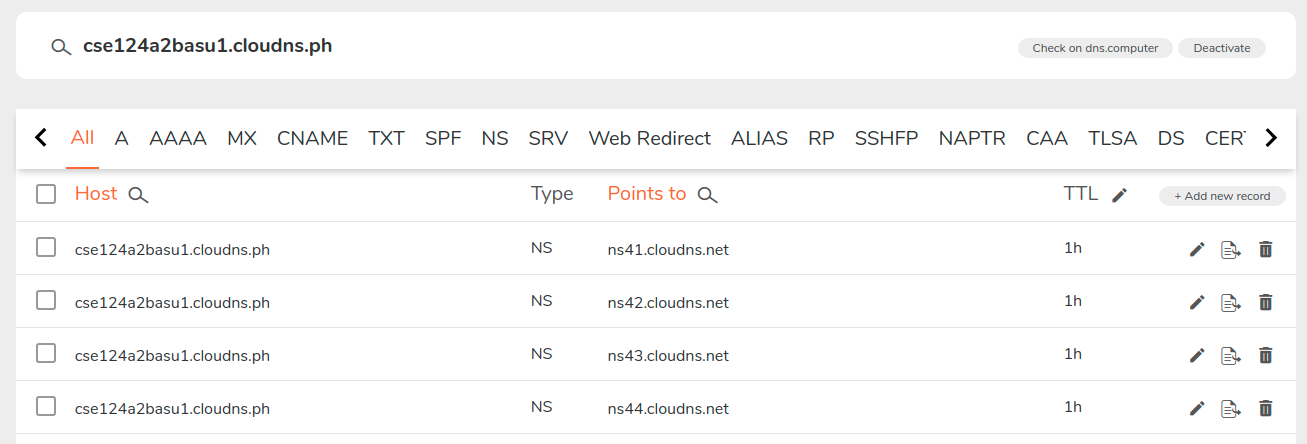
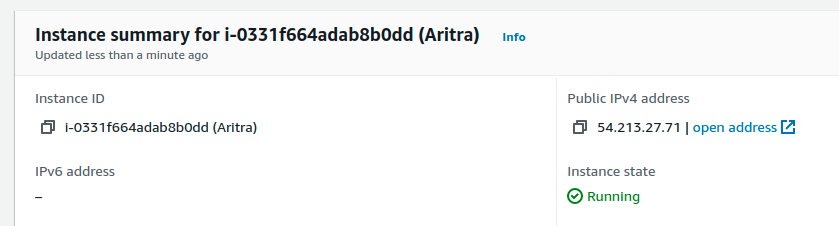
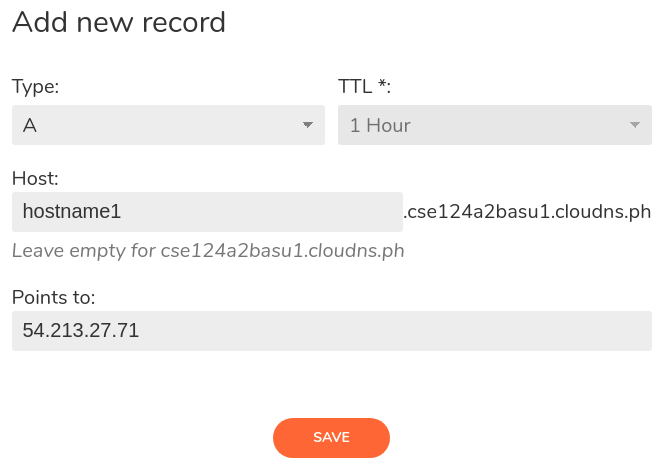
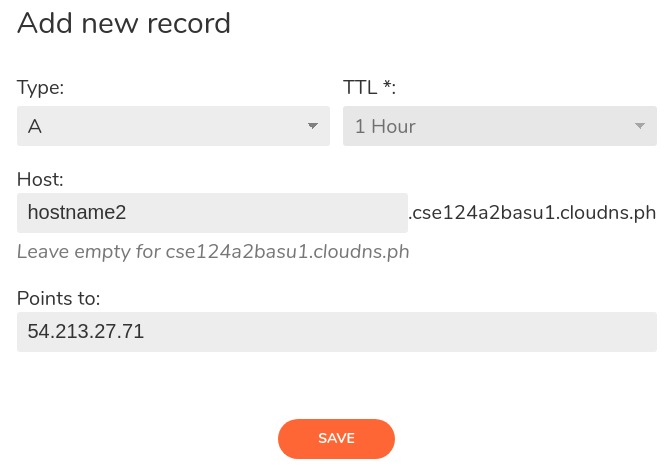
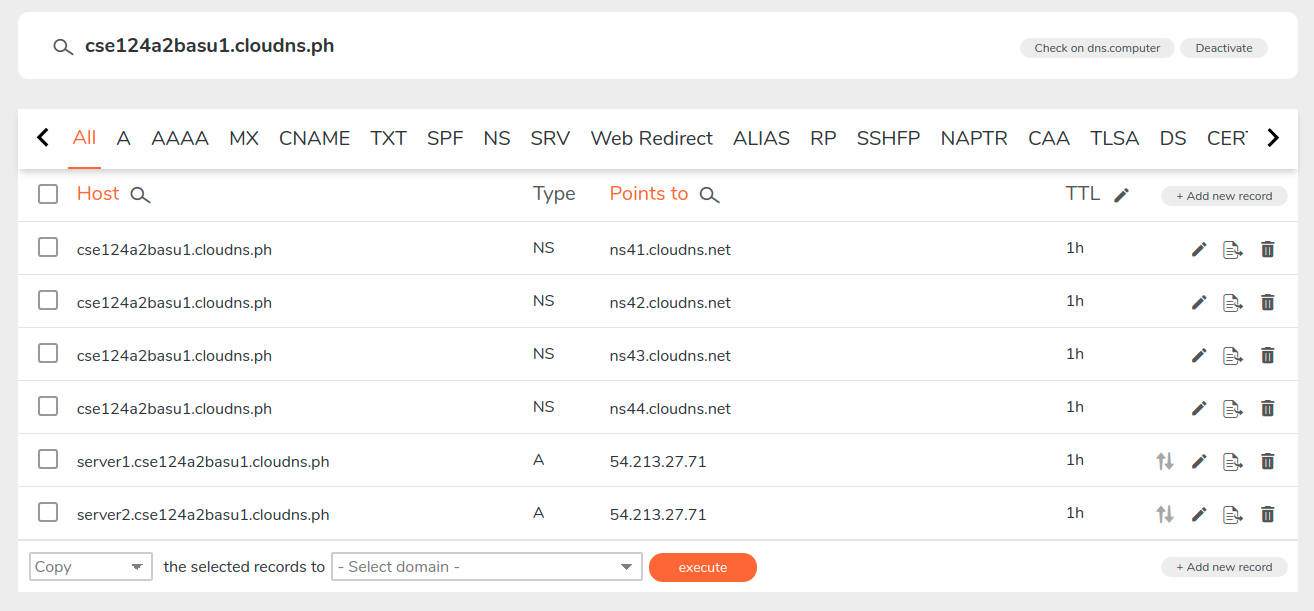
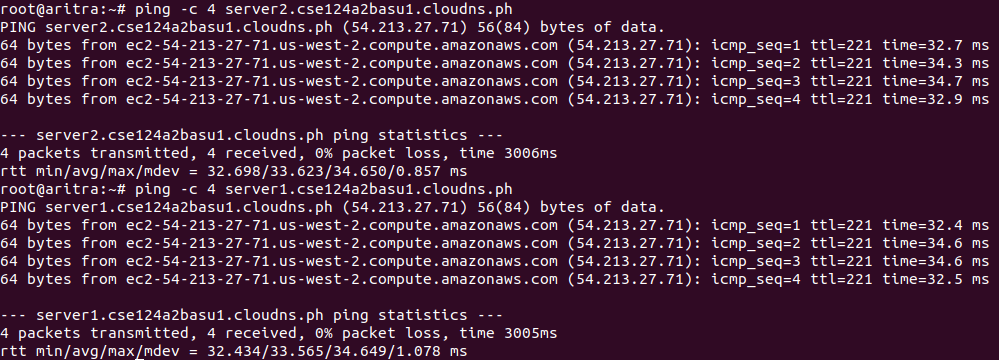
| Content-Type: image/jpeg Content-Type: application/msword Content-Type: text/plain |
| --- |

When serving files out of the document root, your web server will need to convert the file’s extension to one of these “MIME types”. There is a mapping file called mime.types included in your starter code that you should use to determine the mime type for local files so you can properly set the Content-Type response header. If a file is requested with a file extension not included in the mime.types file, you should return “application/octet-stream”.

Do not hard code a path to the mime.types file--use the location specified in the config file.

## Setting up virtual hosting

We will make use of a free third-party web hosting service (<https://www.cloudns.net/>)

* Create a new account.
* Click on the link sent to your mail to activate your account.
* Go to “Dashboard”
* Select “Add new” under the “DNS zones” tab.  
  
* Select “Free zone”.  
  
* “Register” your new free domain after assigning it a valid name (for eg: cse124porter).  
  
* Next, click on your new domain.  
  
* Here, you will find an option to “Add new record”.  
  
* Add 2 different DNS records (with **different hostnames**, but **same server IP** - which will be the **public DNS IPv4 address of your AWS instance**).  
    
    
  
* You can see both the records under your nameserver now.  
  
* Using the “ping” utility command, test to check whether your domain names are being resolved correctly. (Note: It might take a few minutes for the DNS records to be updated after you have saved them).  
  

Once you have registered your domain names and verified that you can “ping” them, you need to update your starter code.

Inside your Git repository, you will see 2 directories - (1) **sample\_htdocs** & (2) **sample\_htdocs\_2** and a JSON file **/src/hosts.json**.

This JSON file will have 2 entries:  
[

{

"hostname": "**<enter\_your\_domain\_name\_here>**",

"docroot": "./sample\_htdocs"

},

{

"hostname": "**<enter\_your\_domain\_name\_here>**",

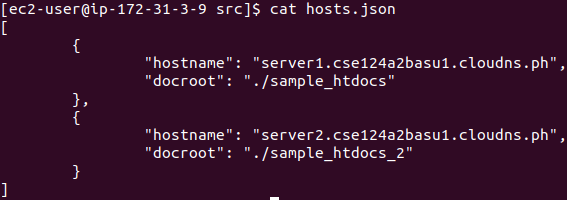
"docroot": "./sample\_htdocs\_2"

}

]

**You need to edit this file. Update the hostname entries with the domain names that you have registered on ClouDNS.** The first entry should index into the files located in the directory sample\_htdocs. The second entry should index into the files located in the directory sample\_htdocs\_2.

For example, this is how the file looks after I added my entries:



## Executable

Your server binary should be called with ./run-server.sh script and should take two arguments which reference two configuration files (which may be stored anywhere on the filesystem, so do not assume it is in the current directory). The first file is the basic configuration, and the second file is the configuration for virtual hosting. For example:

$ ./run-server.sh /home/aturing/myconfig.ini ~aturing/hosts.json

$ ./run-server.sh ~aturing/myconfig.ini /home/aturing/myconfig.ini

$ ./run-server.sh ../configs/myconfig.ini ./hosts.json

# Experimentation

We have provided you with starter code that relies on Go’s own internal http web server. You can use this to experiment and explore with what a “real” webserver does under a variety of conditions. Note that the full HTTP specification is several *thousand* pages long, and we certainly do not expect you to implement all of that! We’re only focusing on a relatively small subset of the overall protocol. As a result, you may see Go’s implementation of the web server doing things we didn’t ask you to (for example, Go’s web server appends a character set specifier to the Content-Type header and returns a header called “X-Content-Type-Options” which we don’t cover in our class. You can just ignore those differences and focus on the subset of HTTP described in the TritonHTTP specification, linked above.

If an aspect of what we’re expecting from you is unclear, please ask. We do encourage you to experiment with Go’s in-built webserver first before asking about whether your web server “should” act in a certain way or not.

# Testing strategies

We have written up a separate document to guide you in developing a testing strategy for your project. This document is located in [Google Drive by clicking here](https://docs.google.com/document/d/1CNscUV-mk6XcaM-19Zs24oGfPOf_csWn96ltYZC8ru4/edit?usp=sharing).

# Grading

Basic functionality for 200 error code responses (30%)

* This category represents error-free, valid requests that result in a 200 status code.
  + The response headers should be set correctly
  + The response body should match the content
* You should support directories and subdirectories
* “http://server:port/" should be mapped to “http://server:port/index.html"
* You should correctly support the MIME types specified in mime.types

Basic functionality for non-200 error code responses (25%):

* Handles 404 for files that aren’t found
* Handles 404 for URLs that escape the doc root
* Correctly handles malformed HTTP requests by issuing a 400 error and closing the connection

Concurrency (5%):

* Your server should be able to handle concurrent clients using goroutines

Pipelining (20%):

* Your server should be able to handle two or more requests that are pipelined together in the same TCP connection
* Your server should handle the Connection: close header correctly
* Your server should implement a 5-second timeout (refreshed after a successful read) correctly for requests that are not explicitly closed via the Connection: close header

Virtual hosting (20%):

* Your server should correctly support virtual domains and virtual hosting

# Submitting your work

Log into gradescope.com and upload your code. This assignment is to be done individually. Make sure to log into gradescope by clicking the link in Canvas! This will ensure that your grade “syncs” correctly. There are two options to submit your code on Gradescope:

* **GitHub**:

Choose your GitHub repository and upload the right branch.

* **Upload**:

Your submission should be a *single ZIP file* named **project2.zip** which should have files with the following structure:

|-- project2.zip

|-- README.md

|-- run-server.sh

|-- src

|-- sample\_htdocs

|-- sample\_htdocs\_2

Make sure that the directory structure remains the same as provided in the starter code.

**Do not include large files(video, audio files) in your submission.**

## **Due date/time**

Indicated on the Canvas site as an assignment