

Assignment 4: Logging and Serving static files

Due: Monday, February 4, 2019

In this assignment you will add logging to your server, and the ability for your server to serve static files (include HTML) to users.

Because these two pieces of functionality are somewhat distinct, it may be useful to have 1-2 members of your team add logging while the rest of the team starts adding static file serving, after which point the teams can converge and add the new logging functionality to the new static file serving code. In other words, you can probably work on these sections in parallel.

1. Add logging to your web server

By now you've probably already added some very basic logging to your server, possibly by writing to `std::cout`. In this assignment you will be expanding on and standardizing this functionality across your server by using a logging framework.

As with previous assignments, we recommend using a Boost library to accomplish this, the [Boost.Log](#) library. You may use another logging library if you really want to, but we've gotten the job done with Boost.Log already so we know it works.

Your server's logging should have this functionality:

- Write to the console, as well as to a file.
- For file logging, a new log file should be started daily at midnight, and when a day's log file size reaches 10 MB. (This is called log file rotation.)
- Log output lines should include, at a minimum, a timestamp, the current thread ID, a severity level, and a message.
- Log basic server messages, such as server startup, config file parsing status, and termination (such as by a signal handler on Ctrl+C).
- Log information about each request that you receive, so you can verify activity in the console as it happens to the server. Requests for the same URL from different computers/clients should be differentiable in the logs.
- Log failure information, such as bad/invalid requests.

When considering what information to log, think about various situations in which you would want to look at log data to analyze your server behavior. (For example, if your server was being hacked, or attacked with a denial-of-service attack, you would probably want to know what IP addresses were hitting your server, so you could block them.) If you can think of a reasonable

scenario that requires a piece of data, then consider logging that data. If you're feeling very uninspired, you could also check out sample logs from other web servers (apache, nginx, etc) and see what other developers/communities/projects have chosen to log.

If you are using Boost.Log, here are some hints:

- Start with trivial logging, i.e. `BOOST_LOG_TRIVIAL`, without any special initialization, to make sure you can get basic console logging working.
- Add `log` as one of the "required components" to your `find_package` call for Boost in `CMakeLists.txt`.
- Add `Boost::log_setup` and `Boost::log` as dependencies (in that order) to your application (i.e. `target_link_libraries` in `CMakeLists.txt`). You will probably get some undefined symbol errors if you omit `log_setup`, or if it is added after `log`.
- It can be hard to tell which header file to include for all the various classes and functions, but the Boost [reference section](#) lists all the top level headers along with what they include, as well as the individual header files (see Utilities section, for example), and their skeletons. If you see a function in documentation and you're not sure where it came from, Ctrl+F search for it in here, and you should be able to find it. If you click on a function name, you should be able to see details (including verification of the header file needed) [like this](#).
- You can accomplish the requirements for this assignment with these functions/macros from Boost.Log:
 - `add_file_log`
 - `add_console_log`
 - `add_common_attributes`
 - `register_simple_formatter_factory`
 - `BOOST_LOG_TRIVIAL`
- The [official tutorial](#) is quite dense beyond the first page, but will give you a pretty comprehensive understanding of the library if you make it all the way through. (Be sure to click "See the complete code" before trying to copy/paste code, the snippets only work in some context.) Also read [this guide](#) for more direct examples of how to set up Boost logging.

You will submit sample log output for 3 consecutive requests, which should originate from more than one computer, as part of your submission to show how what information you are logging.

2. Serve static files from your server

Now that you have a server that can respond to basic browser requests, you will add the ability to serve real files from your server to users.

Importantly, you should maintain the echo functionality in your server, which should be demonstrable for some paths. While you are probably currently echoing requests to any path,

you will now have to handle requests for some paths differently from others. This means you should handle requests differently based on the path being requested. You should develop the concept of a *request handler* that can be given a request object, and generate a response. It's up to you to define the API for this, and in this assignment you should end up with two implementations of a *common request handler interface*, one for handling echo requests, and one for handling file requests. Your two request handlers should inherit from a common base class, defined in a separate header file.

Your server should be made configurable, to serve static files on some path(s) (e.g. `/static`), and serve echo responses on some path(s) (e.g. `/echo`).

Your static file handler should serve files from a configurable *base directory*. For example, if you have a base filesystem directory of `/foo/bar` and define `/static` as a path for static serving, then the URL `http://host:port/static/somefile.html` would serve `/foo/bar/somefile.html` to the user.

Files served should display correctly for common file extensions. For example, `.html` should show up in the browser as a website, and `.jpg` should display as an image. To do this, you need to set the Content-Type HTTP response header properly. Add several common MIME-types that you think would be useful, including at least `.zip` and `.txt` extensions.

If a file isn't present, the static file request handler should return a 404 response code.

Anything configurable should be specified in your config file. Your configuration should be flexible enough to allow the definition of multiple servlets/request handlers of the same type. For example, you could have `http://host:port/static1` and `http://host:port/static2` serve files from different paths, since each request path could be configured with a static file request handler serving from a different base directory.

Make sure your code is adequately unit tested, and consider adding integration test cases. You will probably have to update your integration tests, at the very least.

Your server should run inside the development environment directly, and be able to serve files from there. See the section in the HOWTO on [mapping ports](#) if you have not yet figured out how to expose your development environment server to your local browser.

For deployment on GCP, you will probably need to update your Dockerfile to COPY the static file content into your deployment image.

When you are ready to submit, complete and submit the [form for assignment 4](#).

