Assignment 6

This assignment is about updating your web server to adhere to the new standards we decided on as a class. In a future assignment, students from other teams will be building a new request handler for your web server. In order for them to do this, you must:

- Make your code adhere to the common API
- Write a trivial 404 handler with the common API
- Document how one can contribute a new request handler to your code base

Each student should submit this assignment by 11:59PM on May 16, 2023 into the submission form.

TABLE OF CONTENTS

Refactor your code

Based on the API presentations for Assignment 5, we chose a common API for all teams to use. Now, you must refactor your code to use the common API. As a result of this, you should be able to use a configuration file or request handler written by a different team with your web server.

Configuration files should be able to be shared and parsed with minimal modification, just enough to account for differences in property names. Request handlers could require slight modification, if you have slight differences in field names or header file location. The important thing is to adhere to the spirit and structure of the common API.

Write contributor documentation

Because everyone will be using similar APIs, students from other teams should be able to contribute to your project with a minimal learning curve. Potential contributors need to know things like:

- How the source code is laid out
- How to build, test, and run the code
- How to add a request handler, including:
  - a well-documented example of an existing handler
  - well-documented header files

Create a `README` file with the above instructions on how to work with your code. This should be sufficient for people not intimately familiar with the history of your source code to contribute to it. Ideally they can find all the information they need in the docs without having to ask you any other questions. That's the standard to aim for. (Feel free to link to general CS130 documentation to clarify specific instructions.)

In a future assignment, you may be partially graded on the experience someone has working in your code base, so try to put yourself in a contributor's shoes now and make things as easy as possible for them later!

Write a 404 handler

Write a new request handler using the common API that always returns an HTTP 404 (not found) error code. Configure this handler to `/` so that it handles any request that does not reach one of the other configured handlers.

Grading Criteria

Team grading criteria include those features described above and refactoring according to class-chosen API description below:

- documentation
- a 404 handler

- a common API for all handlers
- a central place that constructs the handlers
- a mechanism to dispatch from an incoming uri to a specific handler

Individual Contributor criteria includes:

- Code submitted for review (follows existing style, readable, testable)
- Addressed and resolved all comments from TL

Tech Lead criteria includes:

- Kept assignment tracker complete and up to date
- Maintained comprehensive meeting notes
- Gave multiple thoughtful (more than just an LGTM) reviews in Gerrit

General criteria includes how well your team follows the class project protocol. Additional criteria may be considered at the discretion of graders or instructors.

Submit your assignment

Everyone should fill out the submission form before 11:59PM on the due date. We will only review code that was submitted to the `main` branch of the team repository before the time of the TL's submission. You may update your submission by re-submitting the form, which will update the submission time considered for grading purposes if you are the TL.

Late hours will acrue based on the submission time, so TL's should avoid re-submitting the form after the due date unless they want to use late hours.

Common API

Based on prior experience in our own server up to this point and after seeing other team's design choices, you each created a proposal and then voted (as a class) to implement the specifications for config file, handler API, and dispatch as noted in class. The decisions there have been summarized below.

Config File Format

- The config uses "#" for comments, which we found convenient because the Nginx parser already supports it.
- Each handler conforms to the "location-major typed" format. The keyword `location`, followed by a serving path, followed by the name of the handler, and a collection of arguments inside `{ ... }`. This format emphasizes the paths, encouraging (but not forcing) their uniqueness.
- Arguments for each handler appear as named elements within the `{ ... }` block. The presence of explicit names avoids ambiguity, aids in readability (especially for humans, but also for machines) and allows for hard-coded default values.
- Each serving path stanza begins with the keyword 'location', that distinguishes from server-level arguments (such as `port`).
- The presence of quoting around strings (e.g. the serving path) is not required.
- Filesystem paths (such as 'root' for the StaticHandler) are relative (implicitly anchored to the webserver binary location).
- Trailing slashes on URL serving paths are optional and should be ignored.

```
port 80; # port my server listens on

location /echo EchoHandler { # no arguments
}

location /static StaticHandler {
  root ./files # supports relative paths
}
```

Request Handlers

- The server has static knowledge of all the possible RequestHandlerFactorys (looks something like a big if-else chain).
- Because the server knows all the types of handlers, a new handler becomes part of the server when via the additional of an `if` clause in the construction block.

```
RequestHandlerFactory* createHandlerFactory(const string& name) {
  if (name == kStaticHandler)
    return new StaticHandlerFactory(...);
  if (name == kEchoHandler)
```

```
    return new EchoHandlerFactory(...);
}
```

- RequestHandlerFactorys have all of their arguments at the time of construction and do *not* have an 'init' method that allows for configuration after construction.
- We decided that RequestHandlerFactorys should be responsible for parsing the Nginx config object, so the server passes the arguments in the `{ ... }` block into the RequestHandlerFactory.

```
// Example constructor signature
public StaticHandler(const string& path, const NginxConfig& config);
```

- RequestHandlerFactorys produce RequestHandlers that have *short* lifetimes, matched to an incoming request. This decision lets us avoid worrying about thread-safe code, as each request can be on its own thread with independent memory (the RequestHandler object is not shared).

Dispatching

- The server dispatches an incoming request by matching the url against a map of paths (built from the config). RequestHandlerFactorys don't participate in this choice.
- The incoming url routes to the handler that has the longest matching prefix.

```
static map<string& location, RequestHandlerFactory*> routes;
string location = match(routes, request->url);
RequestHandlerFactory* factory = routes[location];
RequestHandler* handler = factory->create(location, request->url);
auto status = handler->serve(request, response);
```

- In the spirit of giving more contextual knowledge, the RequestHandler is given both the full path of the requested url and the matching route 'location' path (from the config).

RequestHandler Interface

After deciding which handler to call, the server passes a `boost::beast::http::request` object *and* the address of a `boost::beast::http::response` object for the RequestHandler to fill out its return. A *non-void* return type on the function can pass back error, status, or success information.

```
typedef boolean status; // maybe also an enum or struct with error message?
```

```
class RequestHandler {
 public:
  virtual status handle_request(const request req, response& res) = 0;
}
```

The `boost::beast::http::request` object (documentation) encapsulates the request from the client and contains fields for the url path (`target`), the HTML method (`method`) and version (`version`), headers (`base`) that hold metadata and context, and a body (`body`).

The handler fills out a `boost::beast::http::response` object (documentation) and returns a status indicator to the server. The response object consists of headers that hold metadata associated with the content and session, a body of content, and a code for delivery to the requester.

See here for an example of using the `boost::beast::http` objects.

“You may think using Google’s great, but I still think it’s terrible.” —Larry Page

Page last modified: May 9, 2023.