API Design Proposal

Team: runtime-terror

Authors: Kia Afzali, Victoria Delk, Rohit Ghosh, Yanyu Wang, Charles Zhang

Objective

The major goal of our project is to develop a web server that has robust and well-defined APIs to read and parse the NGINX configurations from config files, echo or respond to HTTP requests, and serve static files. We have divided the project into three major sub-tasks based on the mentioned functionalities, with additional sub-tasks for each, to enhance the readability, maintainability, and extensibility. In the following sections of the proposal, the background of the project, the requirements that we want to achieve, the detailed designs of the project, and alternatives we considered will be discussed to provide a thorough view of the project.

Background

The internet is essentially a collection of webservers with documented APIs that talk to users and other webservers using HTTP requests. The goal of this assignment is for each team to implement a webserver from scratch that can handle various requests and serve static files. While this API is not providing anything useful, the objective of the assignment is to design and implement it using an industry-level approach with an emphasis on testability, expandability, and interpretability. For testing, we are using `gtest` for unit tests and Python scripts for integration tests. We are using polymorphism and dependency injection throughout our code, breaking everything into separate classes. This helps make our code more modular, expandable, and testable. Lastly, we are using a logger to make our program more interpretable, allowing us to track the status of the webserver during development and production. Our webserver is packaged using Docker and is hosted on Google Cloud Platform.

Requirements

The requirements that our design should meet are listed as follows:

- The server is able to configured using a parseable config file
    - The server will return an error message if the config file is invalid
    - The user is able to specify a port number and URL locations for the echo server and static files
- The server is able to echo HTTP requests that are sent to it
    - These are sent in the body of a 200: OK HTTP response
- The server is able to serve static files to clients
    - The server will respond with a 404: Page Not Found response if the URL is not found
    - The server will respond with a 400: Bad Request response if the file at the specified URL is corrupted
    - The server will respond with a 200: OK response if the file at the specified URL is found and not corrupted with the file itself in the message body

Detailed Design

`Location`

```cpp
struct Location
{
    std::string uri;
    std::string handler;
    std::string root;
};
```

- `Location` is an object used to contain route definitions specified in the config

`Request`

```cpp
class Request
{
public:
    Request(const std::string &request);
    std::string ToString() const;
    std::string GetRequestURI() const;

private:
    std::string request_;
};
```

- `Request` is a class used to capture incoming HTTP requests and perform operations on them, constructed using the original request in string format
- The member function `ToString()` can be used to access the underlying HTTP request in string format
- The member function `GetRequestURI()` can be used to extract the URI that the HTTP request was sent to

`Response`

```cpp
class Response
{
public:
    enum Status
    {
        OK = 200,
        BAD = 400,
        NOT_FOUND = 404
    };

    Response(Status status, std::string body);
    void AddHeader(std::string header_name, std::string header_value);
    std::string GetStatusString() const;
    std::string ToString() const;

private:
    struct Header
    {
        std::string name;
        std::string value;

        Header(std::string name, std::string value);
        std::string ToString() const;
    };

    const std::string CRLF = "\r\n";
    Status status_;
    std::vector<Header> headers_;
    std::string body_;
};
```

- `Response` is a class used to construct outgoing HTTP responses from a status code and body content
- `Status` is an enum used to define valid HTTP status codes
- `AddHeader()` is a member function that adds a header name-value pair to the response
- `GetStatusString()` is a member function used to return the status code/descriptor in string format
- `ToString()` is a member function used to return a correctly formatted HTTP response with the information in the `Response` object at the time it is called

Example Config

```
http {
    server {
        listen 80;

        location /echo {
            handler echo;
        }

        location /static1 {
            handler static;
            root /static/static1;
        }

        location /static2 {
            handler static;
            root /static/static2;
        }
    }
}
```

`ConfigReader`

```
class ConfigReader
{
public:
    ConfigReader(NginxConfig *config);
    unsigned short GetPort() const;
    std::vector<Location> GetLocations() const;

private:
    unsigned short ExtractPort(NginxConfig *config) const;
    std::vector<Location> ExtractLocations(NginxConfig *config) const;
    bool ExtractLocation(NginxConfig *location_config, Location *location) const;

    unsigned short port_;
    std::vector<Location> locations_;
};
```

- `ConfigReader` is a class that takes in the parsed config file and is used to access basic config information
- `GetPort()` is a member function that returns the port number that the config file specifies
- `GetLocations()` is a member function that returns a list of all locations defined in the config file

`RequestHandler`

```
class RequestHandler
{
public:
    RequestHandler();
    virtual ~RequestHandler() {}
    virtual Response GenerateResponse(const Request &request) const = 0;
};
```

- `RequestHandler` is the abstract base class from which all types of request handler derive from
- `RequestHandler` defines a pure virtual function `GenerateResponse()` that takes in a `Request` object as a parameter and returns the appropriate `Response` object

`EchoHandler`

```
class EchoHandler : public RequestHandler
{
public:
    EchoHandler();
    Response GenerateResponse(const Request &request) const;
};
```

- `EchoHandler` is a derived class that inherits from `RequestHandler` and implements the echo functionality as defined in the spec
- `EchoHandler` implements the `GenerateResponse()` function by constructing a `Response` object with status code `200`, setting its `Content-Type` header to `text/plain`, setting its `Content-Length` header to the length of the request, and then returning the resulting `Response` object

`StaticHandler`

```
class StaticHandler : public RequestHandler
{
public:
    StaticHandler(const std::string &uri, const std::string &root);
    Response GenerateResponse() const;

private:
    std::string GetMIMEType(const std::string &path) const;
    Response FailedRequest(std::string error) const;
    std::string GetPath() const;

    std::string uri_;
    std::string root_;
};
```

- `StaticHandler` is a derived class that inherits from `RequestHandler` and implements the static file serving functionality as defined in the spec

- `StaticHandler` is constructed from a URI that the handler must respond on and a root directory that the handler serves files from
- `StaticHandler` implements the `GenerateResponse()` function by first constructing the full path of the requested file, then attempting to locate and read the requested file, and finally, setting the proper headers (MIME type, content length, etc.) before returning the constructed `Response` object

`HandlerManager`

```
class HandlerManager
{
public:
    HandlerManager(const std::vector<Location> &locations);
    std::shared_ptr<RequestHandler> GetHandler(const std::string &uri) const;

private:
    std::shared_ptr<RequestHandler> CreateHandler(const Location &location) const;
    void AssignHandler(const Location &location);

    std::unordered_map<std::string, std::shared_ptr<RequestHandler>> handler_map_;
    std::shared_ptr<BadHandler> bad_handler_;
};
```

- `HandlerManager` is a class used to store and access mappings from URIs to handlers, constructed from the list of `Location` objects returned from `ConfigReader`
- `GetHandler()` is a member function that takes in a URI and returns a pointer to the `RequestHandler` associated with it

`Dispatcher`

```
class Dispatcher
{
public:
    Dispatcher(std::unique_ptr<HandlerManager> handler_manager);
    Response DispatchRequest(const Request &request) const;

private:
    std::unique_ptr<HandlerManager> handler_manager_;
};
```

- `Dispatcher` is a class intended to route `Request` objects to the proper `RequestHandler` using a `HandlerManager`
- `DispatchRequest()` is a member function that takes in a `Request` object, and uses `HandlerManager` to access the correct `RequestHandler` and call `GenerateResponse()`

Alternatives Considered

`ConfigReader`

- `GetPort()` and `GetLocations()` originally parsed the config themselves, looking for the relevant keywords, instead of `ConfigReader` doing so on construction
- This original scheme would have seen us passing the `NginxConfig` into `server`/`session` so that a new `ConfigReader` could be constructed whenever we needed any of the config info
- We decided it made more sense for the config to only be needed in `server_main`, and for `ConfigReader` itself to be passed into `server`
- This has the added benefit that if we ever need the port/location data in the future, we won't have to re-parse the config, they will instead already be saved in `ConfigReader`

`Request/Response`

- These classes originally didn't exist, and instead, HTTP requests/responses were passed around in their original string format
- We decided this was too confusing and easy to misuse, so we created objects that would encapsulate this information instead and allow us to define member functions that would make the code more readable

Example Config

- We originally considered defining routes using the block directive from the lecture slides, but decided on this syntax instead because we found it to be more concise and readable

`RequestHandler`

- `RequestHandler` used to store the string format of the HTTP request as a member variable, and then use it to generate the HTTP response
- This required two steps to generate a response: set the request and then generate the response
- Originally, this was done so that the request handler could perform other operations with the request if it needed to, but ultimately, we decided to keep things simple until this functionality was actually needed

`HandlerManager/Dispatcher`

- We considered leaving these classes as a single class, since `HandlerManager` is only ever used as a member variable accessed by `Dispatcher`
- We decided to separate them since we thought their functionality was distinct enough to justify it
- However, having two classes for the dispatching mechanism does increase the complexity significantly