API Design Proposal

Team: server-ihardlyknowher

Authors: Aparna Hariharan, Keyana Desai, Lauren (Missy) Bridgewater, and Natalie Lord

Objective

Over the past five weeks, our team has developed an API for a configurable web server to handle and respond to a variety of client requests. The major goals of this project include enabling server configuration through an Nginx-style file and correctly handling a wide range of HTTP/1.1 requests, including echo and static file responses. Our API is designed to be easily extendable, allowing us to add new handlers by specifying their URIs and relevant parameters directly in the configuration file. This approach addresses our non-goal of hardcoded routing logic, which is difficult to maintain and scale. By prioritizing flexibility, our design enables the server to support more complex functionality without requiring major code changes.

Background

As our web server grows in complexity, we need a consistent and expandable way to add new functionality. In order to handle a variety of HTTP/1.1 client requests, we developed a packet structure for requests and responses supported by helper functions in res_req_helpers.cc. When a raw request is received from the client, it is parsed to create a request packet with a header and a body. Similarly, outgoing responses are serialized into properly formatted HTTP responses by converting the http_version, status_code, reason_phrase, and response body into a stream. This response and request API provides a consistent and expandable interface for request processing across various handlers.

This API design is integrated into a single server session loop in session.cc, which listens to incoming data, parses it into a request, and delegates it to the correct handler based on longest matching URI. Currently, we support two handlers: echo_handler.cc, which returns the request body unchanged, and static_file_handler.cc, which attempts to locate and serve files based on a mount point-to-document root mapping defined in the dkr_config configuration file. Both handlers inherit from a base class, request_handler.h, which defines a virtual method handle_request that takes in a request structure and returns a response structure. The method handle_request is then implemented in the children classes, EchoHandler and StaticFileHandler.

In this method, the static file handler returns a 404 not found error if the requested URI does not exist or is malformed. If the requested URI is found, the method returns a response packet with the response body including the contents of the file requested. In the EchoHandler class, handle_request returns a response packet containing the request method, uri, and http_version. These handlers are then created in the main function defined in server_main.cc and passed into each session that occurs. This modular handler architecture is designed to scale with additional functionality by simply defining new handler classes.

Requirements

Our web server is designed to meet several key requirements. It must be configurable through a file that specifies the port number, maps URIs to request handlers, and defines settings such as the document root for serving static files. The server must support multiple request handlers, like EchoHandler and StaticFileHandler, through a common interface and correctly parse HTTP/1.1 GET requests. It should return correct responses, including 200 OK for valid requests and 404 Not Found for missing or invalid requests. Static files must include accurate content types for proper browser rendering. Additionally the server must support logging to both the console and a file, with log rotation at midnight or after 10MB. Each log entry should include a timestamp, security level, thread ID, and message.

The main goals of our design are extensibility, modularity, and usability. To support extension, a new handler can be added through writing a class that inherits from request_handler.cc and updating the configuration file, without having to change the server code. Additionally, the code should have modular components for parsing, dispatching, handling requests, and logging, so that each part of the code can be tested and updated individually. Specifically, the dispatching and handling of requests must be as efficient as possible. The design should emphasize readable and consistent code, such as keeping main minimal and delegating responsibilities to dedicated classes. To monitor and understand server behavior, the server should output useful, non-redundant, and concise logs. Also, we have focused on ensuring our server is consistent across different environments, including local machines, Docker containers, and Google Cloud.

There are certain features we decided to exclude from our design. Since our server handles HTTP/1.1, we do not support HTTPS or TLS encryption. Additionally, our server does not handle concurrent requests, it processes them sequentially. By not implementing these features, we keep our project simple and focused on core functionality. We also avoid hardcoded routing logic, which is difficult to extend as more handlers are needed.

Detailed design

1. Overview of Server Flow

Upon startup, our server reads from a configuration file to determine the port number and initialize the proper request handlers. For each incoming client connection, it parses the request, dispatches it to the correct handler based on the URI, and returns the generated response.

2. Configuration Processing

2.1 Nginx-style Parsing

The configuration file (e.g., `dkr_config`) is parsed using a recursive descent parser implemented in `nginx_config_parser.cc`. This handles block nesting, quoted strings, and tokenization. A brief Nginx-style configuration file example is shown below.

```
listen 80;
location /static/ {
  root /app/static;
}
```

2.2 Config Interpretation

The parser generates an in-memory `NginxConfig` tree. `config_interpreter.cc` then extracts the port number via `find_listen_port()` and a map of mount points to doc roots via `extract_location_root_mappings()`. An example result can be seen below.

```
{
  "/static/"  -> "/app/static",
  "/images/"  -> "/app/images"
}
```

These mappings are then used to instantiate the correct handlers in `main()`.

3. Request Handler Architecture

3.1 Base Interface

All handlers inherit from `RequestHandler`, which defines the following interface.

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

Currently, two handlers are defined: `EchoHandler` and `StaticFileHandler`. Handlers are stored as a map of URIs to a shared pointer to the request handler.

```
std::map<std::string, std::shared_ptr<RequestHandler>> uri_to_handler;
```

In `main()`, we initialize an echo handler and use the map of mount points to doc roots to create the static file handlers.

```
handlers["/echo"] = std::make_shared<EchoHandler>();
for (const auto& pair : mount_to_docroot) {
  handlers[pair.first] = std::make_shared<StaticFileHandler>(pair.first,
pair.second);
}
```

3.2 EchoHandler

The `EchoHandler` returns the incoming request as the response body.

3.3 StaticFileHandler

The `StaticFileHandler` maps URIs, like `/static/file.txt`, to corresponding full file system paths, like `/app/static/file.txt`. It serves the requested file contents if the file exists, or otherwise returns a 404 Not Found error.

4. Server and Session Lifecycle

4.1 Server Initialization

`server.cc` creates a TCP acceptor using Boost.Asio that listens on the port extracted from the configuration. When a new client connects, a new `session` instance is created, and its socket is passed to begin communication.

4.2 Session Handling

Each connection is handled by a `session` object, which asynchronously reads data, parses it into a `request`, and dispatches it to the longest-prefix matching handler. If no handler matches, the EchoHandler is used as a fallback. The resulting `response` is serialized and sent back to the client before the session shuts down. The logic follows the following:

```
request req = parse_request(request_buffer_);
auto handler = find_best_prefix_match(req.uri);
response res = handler->handle_request(req);
auto out = serialize_response(res);
boost::asio::async_write(socket_, boost::asio::buffer(out), ...);
```

5. Request and Response API

Defined in `res_req_helpers.cc`, this struct-based API allows consistent request parsing and response generation.

5.1 Request Struct

```
struct request {
  std::string method;
  std::string uri;
  std::string http_version;
  std::map<std::string, std::string> headers;
  std::string body;
};
```

5.2 Response Struct

```
struct response {
  std::string http_version;
  int status_code;
  std::string reason_phrase;
  std::map<std::string, std::string> headers;
  std::string body;
};
```

5.3 Usage in Handlers

The EchoHandler echos back the request.

```
response EchoHandler::handle_request(const request& req) {
  \\ ...
```

```
  return {"HTTP/1.1", 200, "OK", {}, req.body};
}
```

The static file handler displays the contents of the file in the browser.

```
response StaticFileHandler::handle_request(const request& req) {
  \\ ...
  resp.status_code = 200;
  resp.reason_phrase = "OK";
  resp.headers["Content-Type"] = mime;
  resp.headers["Content-Length"] = std::to_string(data.size());
  resp.body = std::move(data);
  return resp;
}
```

6. Logging and Observability

Implemented in `logger.cc` using Boost.Log:

- Logs to both console and file
- Daily or 10MB rotation
- Output format includes timestamp, severity, and thread ID

Example:

```
[2025-05-04 18:34:12] [info] Server started, listening on port 80
```

7. Extensibility Patterns

To add a new handler:

- Create a class inheriting from `RequestHandler`
- Implement `handle_request()`
- Register in the config file (future enhancement)

Because handler routing is dynamic and not hardcoded, the server is open to adding features like `ImageHandler`, `HealthCheckHandler`, or even `ReverseProxyHandler` with minimal disruption.

Alternatives considered

One alternative design choice was the location of handler instantiation. Initially, we created new handler instances for each session, meaning that new objects were unnecessarily created every time a new client connected. Instead, we chose to create each handler in the main function and pass them as shared pointers to each server and session. By reusing handlers, our code is more efficient and allows handlers to be reused across sessions. The following two alternate designs are designs we plan to include after watching our classmates' API presentations and recognizing some inconsistencies between our goals and implementation. Originally, we chose to hardcode EchoHandler directly in the server instead of specifying it in the configuration file. While this works, it is inconsistent with how StaticFileHandler is set up and would make adding handlers more difficult in the future. To support consistency, we plan to move the EchoHandler definition into the config file, just like the other handlers. Also, we currently use a linear search to find the best-matching URI prefix for incoming requests, but plan to implement a TRIE search instead. Our first design would make dispatching requests slower as the number of request handlers scaled. By using a TRIE search, we meet the goal of keeping the dispatch of requests efficient while keeping the same functionality.