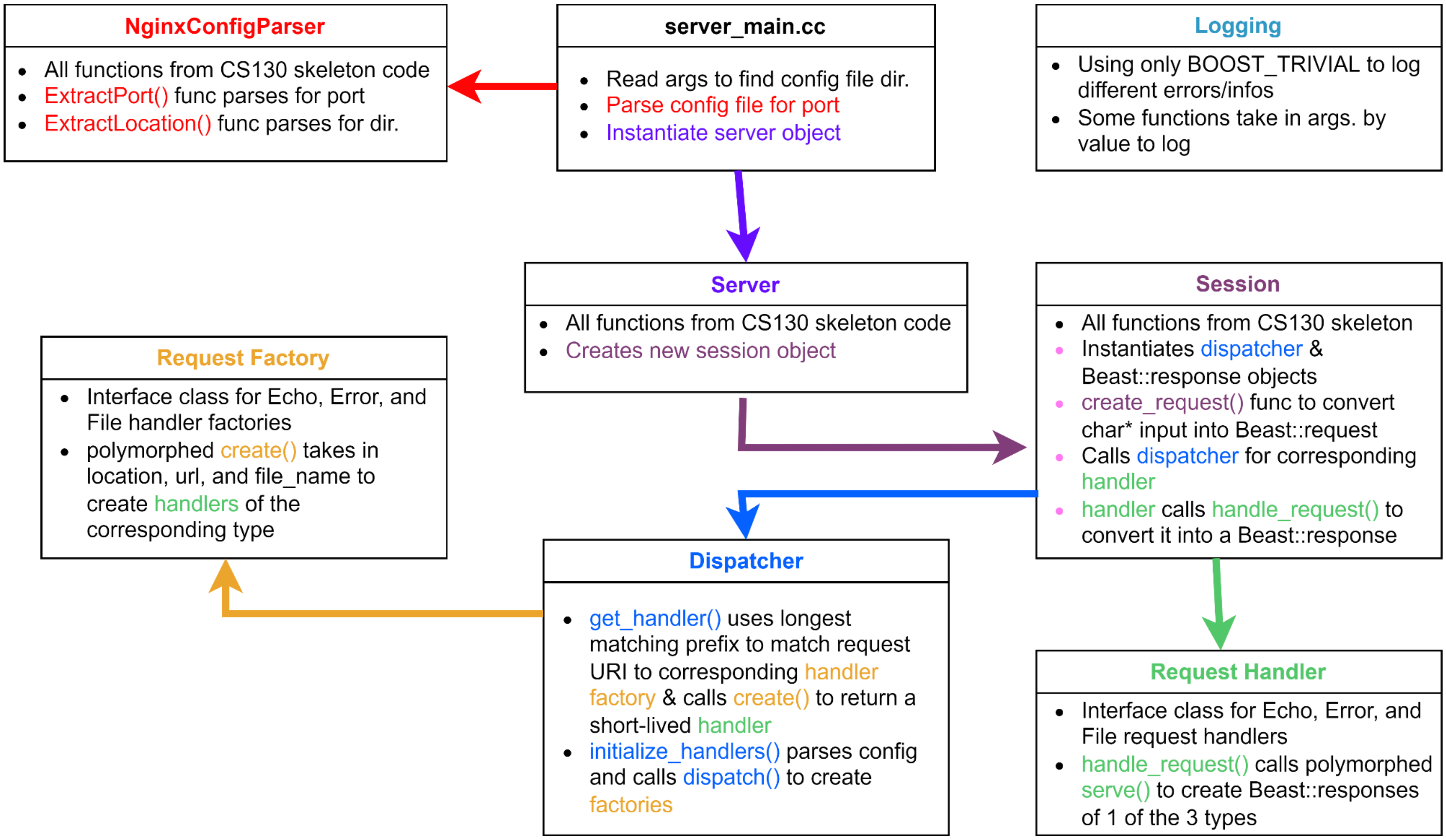

NginxConfigParser
All functions from CS130 skeleton code
ExtractPort() func parses for port
ExtractLocation() func parses for dir.
server_main.cc
Read args to find config file dir.
Parse config file for port
Instantiate server object
Logging
Using only BOOST_TRIVIAL to log different errors/infos
Some functions take in args. by value to log
Server
All functions from CS130 skeleton code
Creates new session object
Session
All functions from CS130 skeleton
Instantiates dispatcher & Beast::response objects
create_request() func to convert char* input into Beast::request
Calls dispatcher for corresponding handler
handler calls handle_request() to convert it into a Beast::response
Request Factory
Interface class for Echo, Error, and File handler factories
polymorphed create() takes in location, url, and file_name to create handlers of the corresponding type
Dispatcher
get_handler() uses longest matching prefix to match request URI to corresponding handler factory & calls create() to return a short-lived handler
initialize_handlers() parses config and calls dispatch() to create factories
Request Handler
Interface class for Echo, Error, and File request handlers
handle_request() calls polymorphed serve() to create Beast::responses of 1 of the 3 types

1 Introduction and High-level Overview

This diagram is a high-level view of the relationships and purpose of the classes in our implementation. Further on this document, we will explain the source code in more detail. In short, our server object is created in the server_main.cc file. The server creates a session object which in turn uses the dispatcher to call handler factories to create a short-lived request handler which will serve the response.

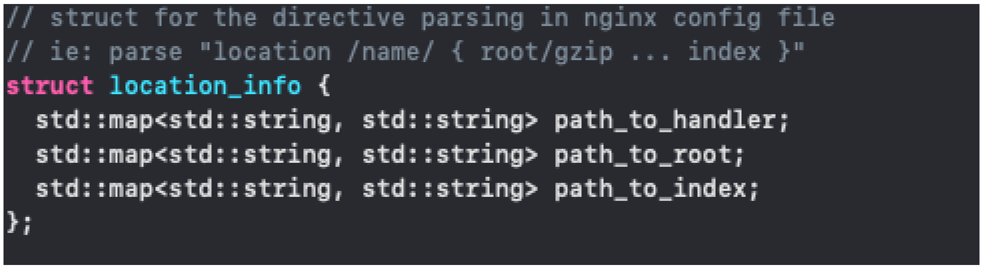
To run the server:

1. Pull the latest commit from our sffs project main branch.
2. Create a build directory inside the sffs directory and run **cmake ..** inside the build folder
3. Run **make** and wait for the build to complete
4. Run **bin/server ../real_config** to test locally
5. In another terminal, send http requests to **localhost 8080** to test for responses

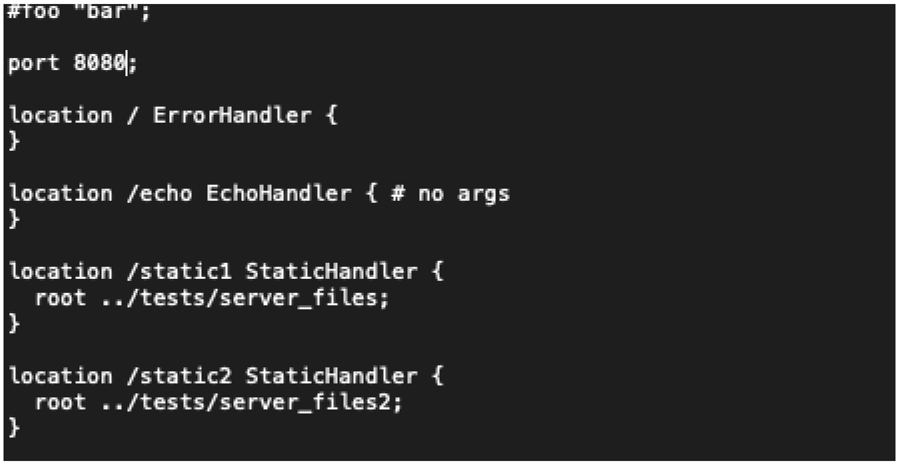
2 Explanation of Source Code

2.1 Config Parser

The config parser extracts the port number by identifying the tokens “listen” or “port” inside the file and get the numbers right after it (the ExtractPort function that returns the integer for the port). Once the port number is validated, the parser can be used to extract the location blocks that have the information for the request handlers the server’s session needs to initialize (the ExtractLocation function that returns maps from path to handler and from path to the root directory that server uses for the request directory mapping).



```
// struct for the directive parsing in nginx config file
// ie: parse "location /name/ { root/gzip ... index }"
struct location_info {
  std::map<std::string, std::string> path_to_handler;
  std::map<std::string, std::string> path_to_root;
  std::map<std::string, std::string> path_to_index;
};
```



```
#foo "bar";

port 8080;

location / ErrorHandler {
}

location /echo EchoHandler { # no args
}

location /static1 StaticHandler {
  root ../tests/server_files;
}

location /static2 StaticHandler {
  root ../tests/server_files2;
}
```

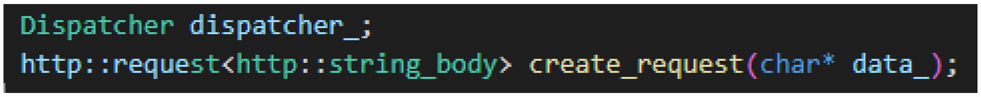
For example, path_to_handler[“/echo”] == ErroHandler,
path_to_root[“/static1”] == ../tests/server_files

2.2 Server

The Server class code is largely unchanged from the initial skeleton code provided by the course.

2.3 Session

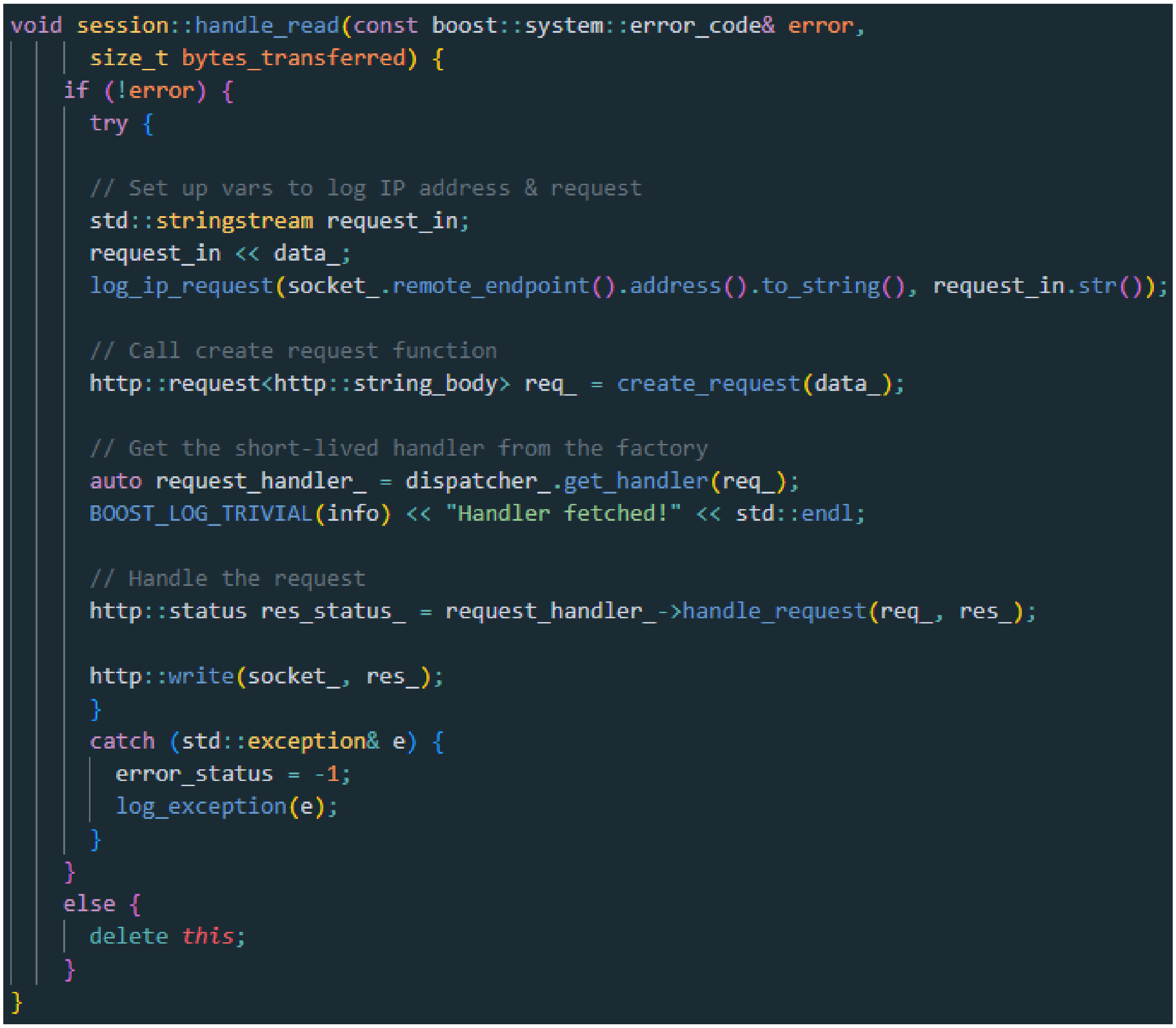
Our Session class now holds two additional private variables:



```
Dispatcher dispatcher_;
http::request<http::string_body> create_request(char* data_);
```

The dispatcher is created with every session and manages the creation & usage of different RequestHandlerFactory objects. In addition, Session uses **create_request()** to convert the string/char array input from the client into an appropriate Beast::request object.

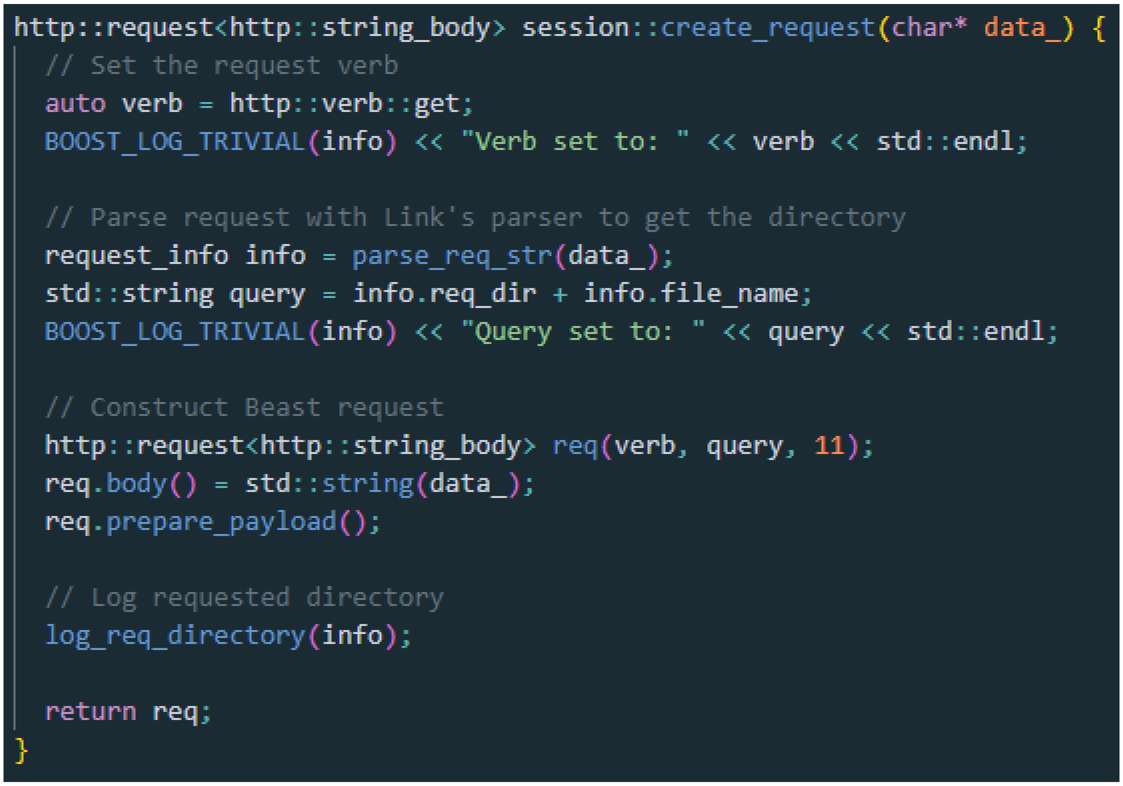
A majority of the changes are in **Session::handle_read()** which now notably does logging & calls the dispatcher to get the matching handler.



```
void session::handle_read(const boost::system::error_code& error,
      size_t bytes_transferred) {
    if (!error) {
      try {

        // Set up vars to log IP address & request
        std::stringstream request_in;
        request_in << data_;
        log_ip_request(socket_.remote_endpoint().address().to_string(), request_in.str());

        // Call create request function
        http::request<http::string_body> req_ = create_request(data_);

        // Get the short-lived handler from the factory
        auto request_handler_ = dispatcher_.get_handler(req_);
        BOOST_LOG_TRIVIAL(info) << "Handler fetched!" << std::endl;

        // Handle the request
        http::status res_status_ = request_handler_->handle_request(req_, res_);

        http::write(socket_, res_);
      }
      catch (std::exception& e) {
        error_status = -1;
        log_exception(e);
      }
    }
    else {
      delete this;
    }
}
```

The short-lived handler **request_handler_** will call its handle-request() function, taking the Beast::request & response objects and returning an http::status object that we have yet to utilize. Ideas for utilizing this may include a final error checking before the response is written to the client using **http::write()**.

Session uses **create_request()** along with a string request parser to generate a **request_info** struct containing multiple strings corresponding to the requested directory, the base directory, and the file_name.

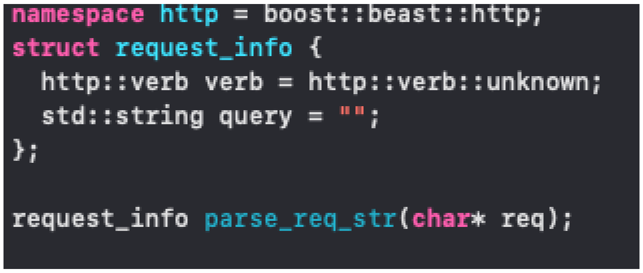


```cpp
http::request<http::string_body> session::create_request(char* data_) {
  // Set the request verb
  auto verb = http::verb::get;
  BOOST_LOG_TRIVIAL(info) << "Verb set to: " << verb << std::endl;

  // Parse request with Link's parser to get the directory
  request_info info = parse_req_str(data_);
  std::string query = info.req_dir + info.file_name;
  BOOST_LOG_TRIVIAL(info) << "Query set to: " << query << std::endl;

  // Construct Beast request
  http::request<http::string_body> req(verb, query, 11);
  req.body() = std::string(data_);
  req.prepare_payload();

  // Log requested directory
  log_req_directory(info);

  return req;
}
```

It then creates the request object by setting the request's verb, query, and HTTP version (1.1) and calls prepare_payload() to set the **Content-Length**.

2.3.a Request Parser

The request parser is a simple struct with a parsing function **parse_req_str()**.



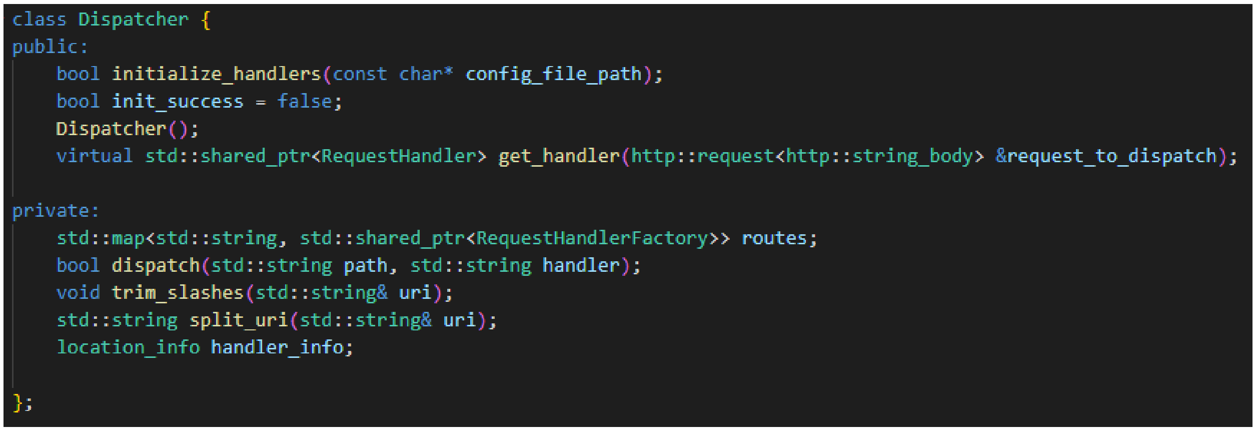
```cpp
namespace http = boost::beast::http;
struct request_info {
  http::verb verb = http::verb::unknown;
  std::string query = "";
};

request_info parse_req_str(char* req);
```

The request_info struct's purpose is to hold the input from the client to later convert into the Beast::request.

The parse_req_str function will extract the verb of the request like "get", "post", etc. If it's not valid, it will be set to unknown for the beast::http::verb value.

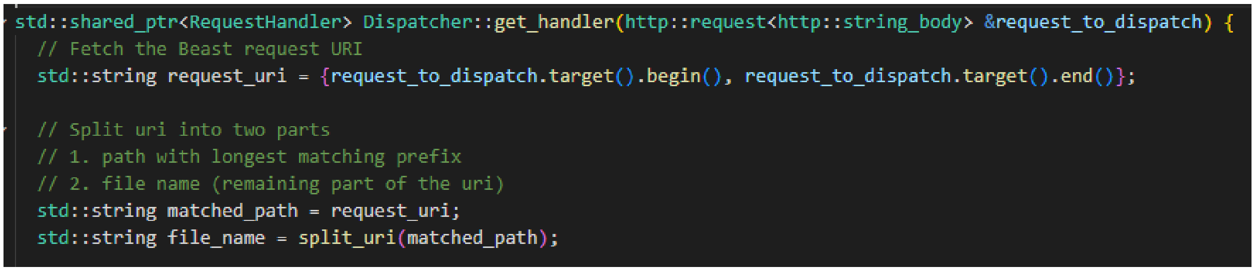
It will also extract the query part of the request like the file path of the request "/static1/index.html" for later use inside the dispatcher.

2.4 Dispatcher

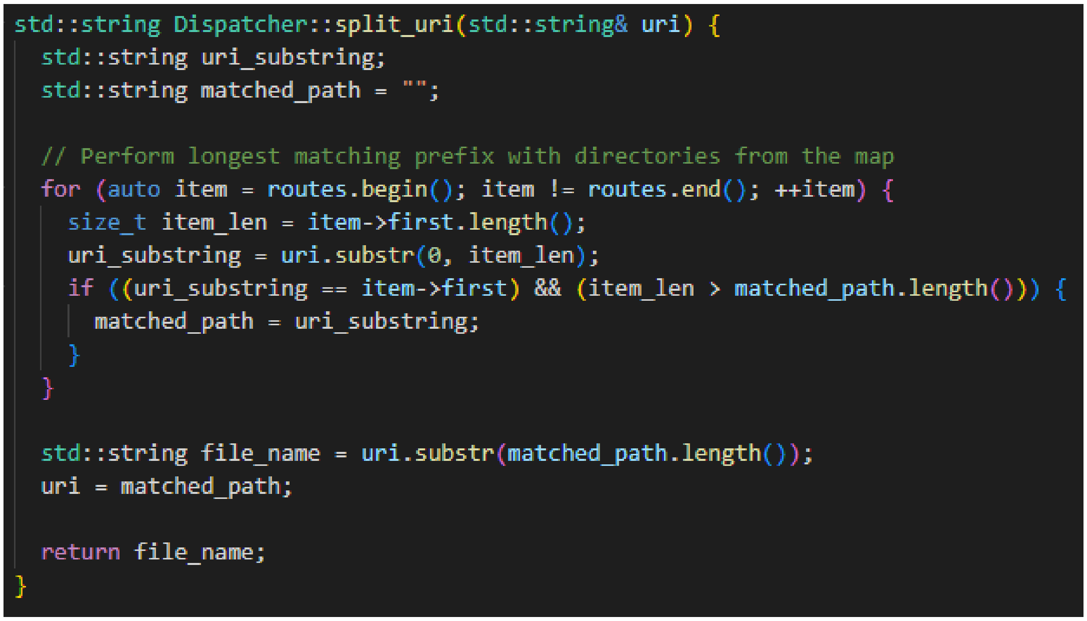


```
class Dispatcher {
public:
    bool initialize_handlers(const char* config_file_path);
    bool init_success = false;
    Dispatcher();
    virtual std::shared_ptr<RequestHandler> get_handler(http::request<http::string_body> &request_to_dispatch);

private:
    std::map<std::string, std::shared_ptr<RequestHandlerFactory>> routes;
    bool dispatch(std::string path, std::string handler);
    void trim_slashes(std::string& uri);
    std::string split_uri(std::string& uri);
    location_info handler_info;

};
```

The primary function of the Dispatcher class is to take a Beast::request and match the corresponding URI with the existing directories that were parsed on start-up. The **get_handler()** function extracts the **request_uri** and calls **split_uri()** to find the longest matching prefix and use the rest of the uri as the file name.

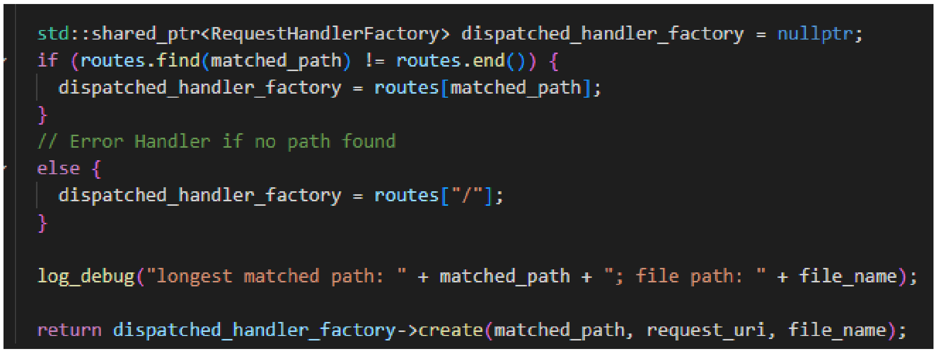


```
std::shared_ptr<RequestHandler> Dispatcher::get_handler(http::request<http::string_body> &request_to_dispatch) {
  // Fetch the Beast request URI
  std::string request_uri = {request_to_dispatch.target().begin(), request_to_dispatch.target().end()};

  // Split uri into two parts
  // 1. path with longest matching prefix
  // 2. file name (remaining part of the uri)
  std::string matched_path = request_uri;
  std::string file_name = split_uri(matched_path);
```



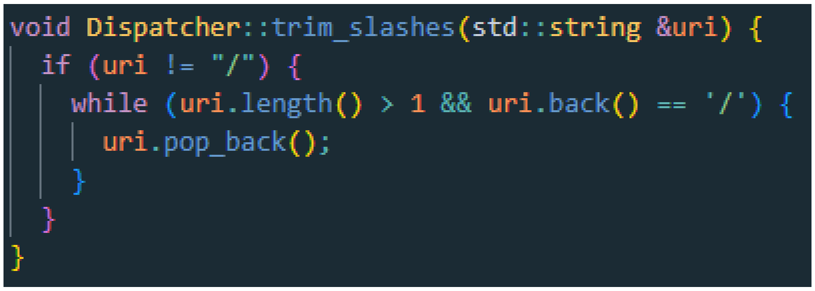
```
std::string Dispatcher::split_uri(std::string& uri) {
  std::string uri_substring;
  std::string matched_path = "";

  // Perform longest matching prefix with directories from the map
  for (auto item = routes.begin(); item != routes.end(); ++item) {
    size_t item_len = item->first.length();
    uri_substring = uri.substr(0, item_len);
    if ((uri_substring == item->first) && (item_len > matched_path.length())) {
      matched_path = uri_substring;
    }
  }

  std::string file_name = uri.substr(matched_path.length());
  uri = matched_path;

  return file_name;
}
```

After getting the longest matching prefix, it gets the appropriate **RequestHandlerFactory**. Using this factory, it will call a polymorphed version of **RequestHandlerFactory::create()** to return either a short-lived Echo, Error, or File request handler.



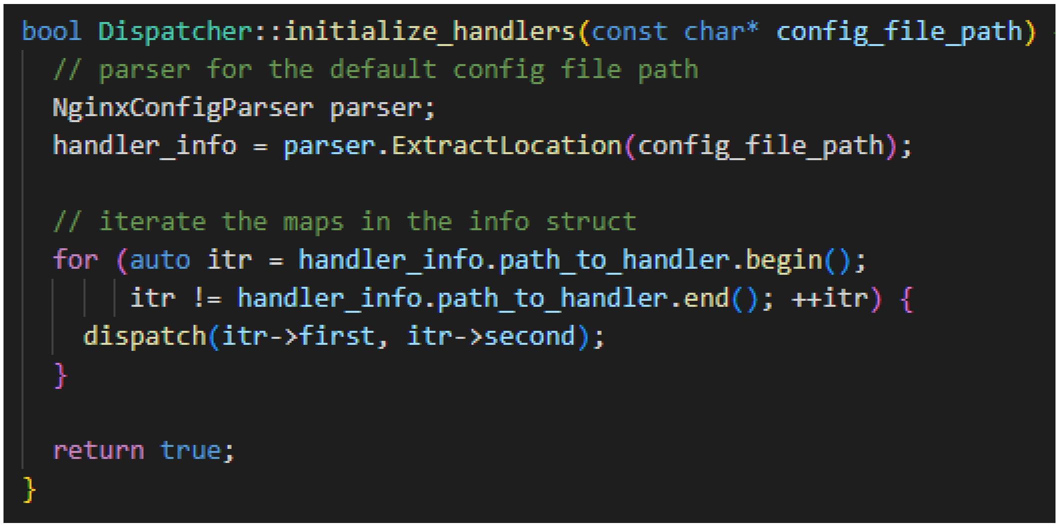
```
std::shared_ptr<RequestHandlerFactory> dispatched_handler_factory = nullptr;
if (routes.find(matched_path) != routes.end()) {
  dispatched_handler_factory = routes[matched_path];
}
// Error Handler if no path found
else {
  dispatched_handler_factory = routes["/"];
}

log_debug("longest matched path: " + matched_path + "; file path: " + file_name);

return dispatched_handler_factory->create(matched_path, request_uri, file_name);
```

In order to tackle the trailing slashes issue, we created a function which simply removes all the trailing slashes from the end of the string (regardless of how many trailing slashes there are////////////////).



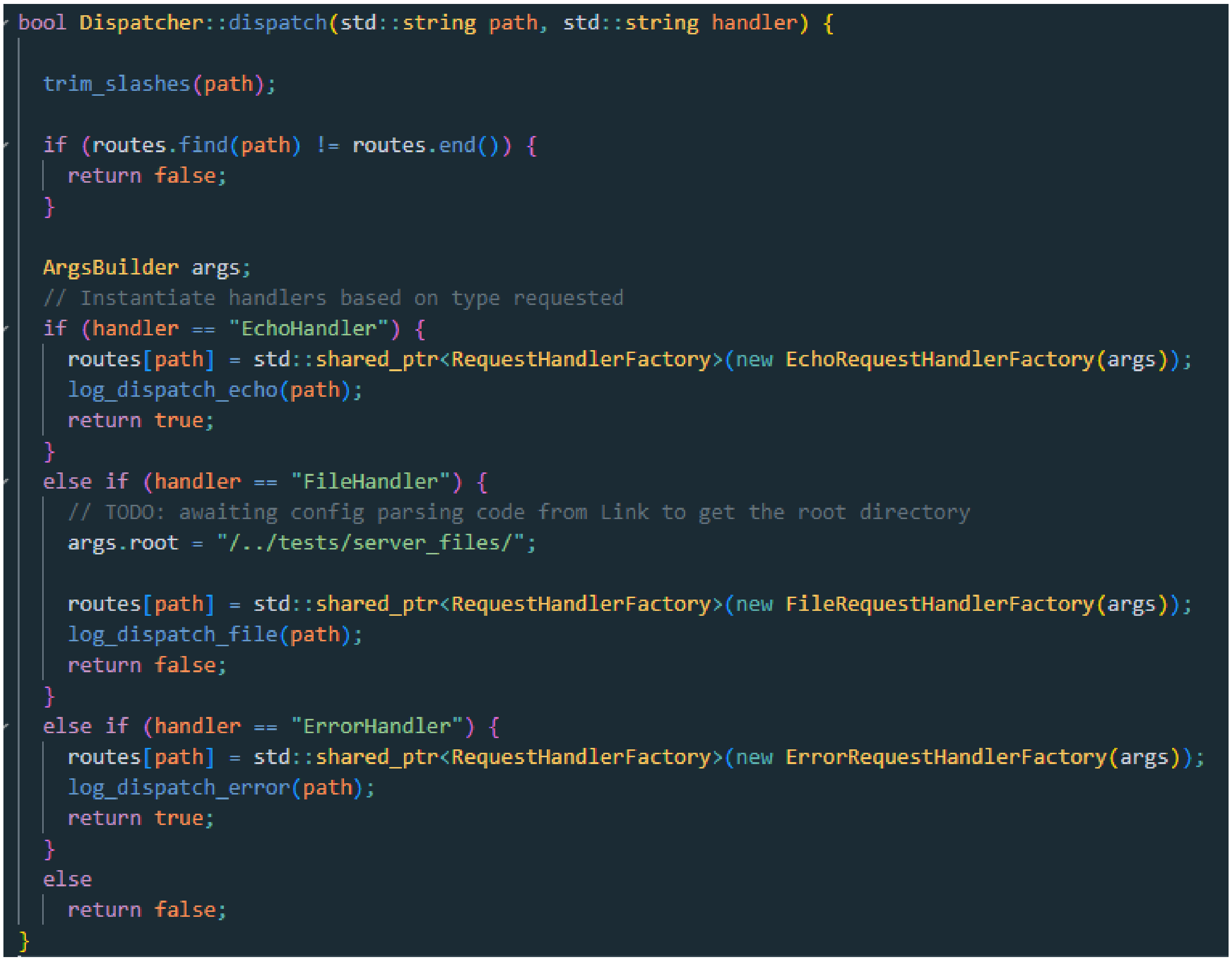
```
void Dispatcher::trim_slashes(std::string &uri) {
  if (uri != "/") {
    while (uri.length() > 1 && uri.back() == '/') {
      uri.pop_back();
    }
  }
}
```

initialize_handlers() is a public method that takes in the path to our config file and creates an **NginxConfigParser** object to parse it. After extracting the config information, it calls **dispatch()** to create a map of paths to request handler factories.



```
bool Dispatcher::initialize_handlers(const char* config_file_path)
  // parser for the default config file path
  NginxConfigParser parser;
  handler_info = parser.ExtractLocation(config_file_path);

  // iterate the maps in the info struct
  for (auto itr = handler_info.path_to_handler.begin();
        itr != handler_info.path_to_handler.end(); ++itr) {
    dispatch(itr->first, itr->second);
  }

  return true;
}
```

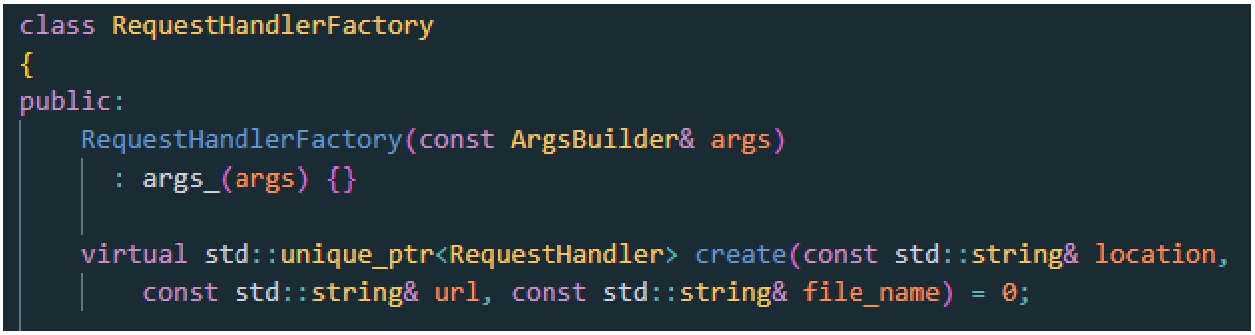
The **dispatch()** private method of the Dispatcher class takes the parsed paths and handler type and fills its route map object with the corresponding path and a request handler factory. The boolean return value of the function allows us to potentially implement error checks in future implementations.



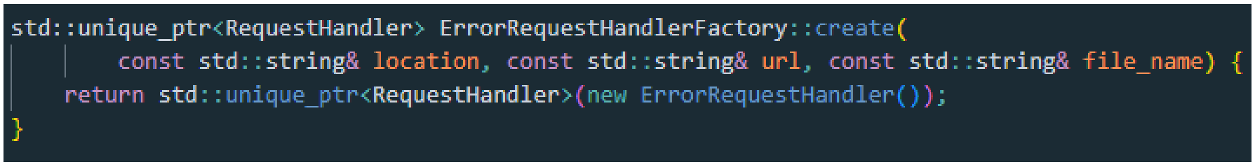
```
bool Dispatcher::dispatch(std::string path, std::string handler) {

  trim_slashes(path);

  if (routes.find(path) != routes.end()) {
    return false;
  }

  ArgsBuilder args;
  // Instantiate handlers based on type requested
  if (handler == "EchoHandler") {
    routes[path] = std::shared_ptr<RequestHandlerFactory>(new EchoRequestHandlerFactory(args));
    log_dispatch_echo(path);
    return true;
  }
  else if (handler == "FileHandler") {
    // TODO: awaiting config parsing code from Link to get the root directory
    args.root = "/../tests/server_files/";

    routes[path] = std::shared_ptr<RequestHandlerFactory>(new FileRequestHandlerFactory(args));
    log_dispatch_file(path);
    return false;
  }
  else if (handler == "ErrorHandler") {
    routes[path] = std::shared_ptr<RequestHandlerFactory>(new ErrorRequestHandlerFactory(args));
    log_dispatch_error(path);
    return true;
  }
  else
    return false;
}
```

2.5 Request Handler Factory

The RequestHandlerFactory interface has a virtual function which is polymorphed by each of the 3 request types. In all cases, the function takes in a string location, url, and file_name as part of being given “full context” as mentioned in the Assignment 6 specification.



```
class RequestHandlerFactory
{
public:
    RequestHandlerFactory(const ArgsBuilder& args)
      : args_(args) {}

    virtual std::unique_ptr<RequestHandler> create(const std::string& location,
        const std::string& url, const std::string& file_name) = 0;
```

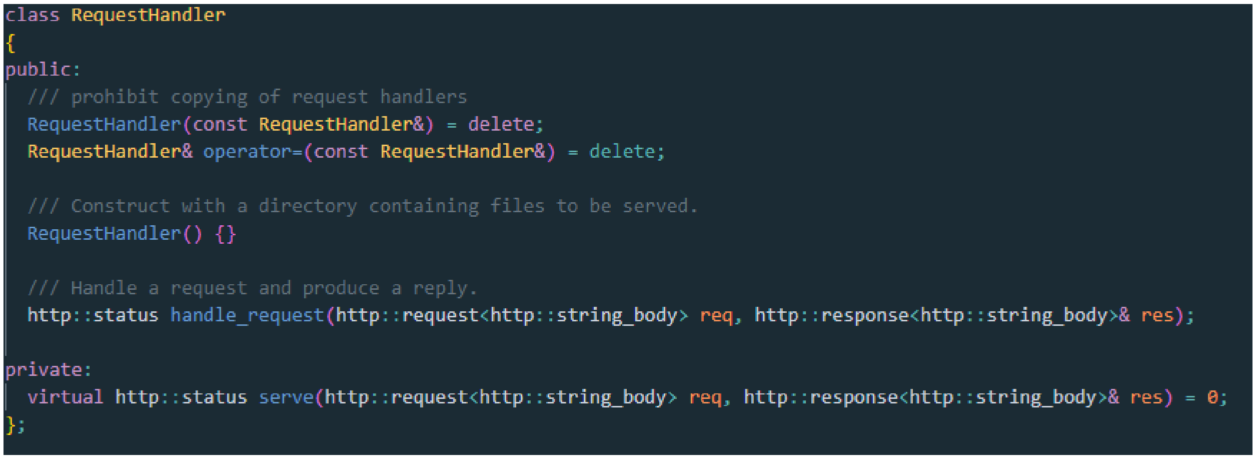


```
std::unique_ptr<RequestHandler> ErrorRequestHandlerFactory::create(
        const std::string& location, const std::string& url, const std::string& file_name) {
    return std::unique_ptr<RequestHandler>(new ErrorRequestHandler());
}
```

Since each RequestHandler corresponds to exactly one request, we used std::unique_ptr to make it clear that multiple pointers to the same object would not exist.

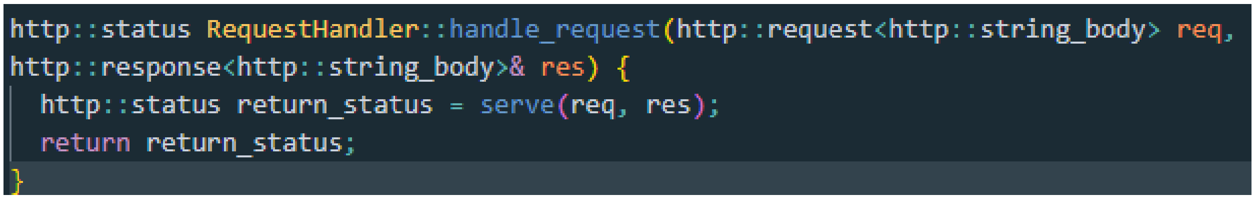
2.6 Request Handler

The RequestHandler class is an interface for the 3 child handlers, Echo, Error, and File.



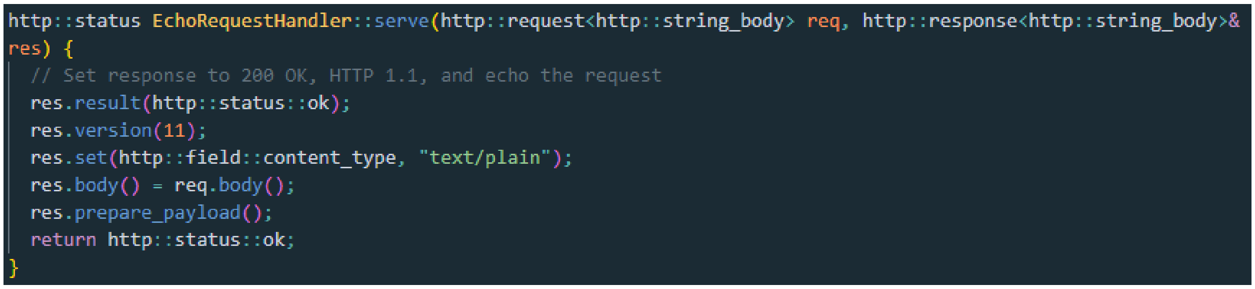
```
class RequestHandler
{
public:
  /// prohibit copying of request handlers
  RequestHandler(const RequestHandler&) = delete;
  RequestHandler& operator=(const RequestHandler&) = delete;

  /// Construct with a directory containing files to be served.
  RequestHandler() {}

  /// Handle a request and produce a reply.
  http::status handle_request(http::request<http::string_body> req, http::response<http::string_body>& res);

private:
  virtual http::status serve(http::request<http::string_body> req, http::response<http::string_body>& res) = 0;
};
```

The parent class has a **handle_request()** function that takes in both a request and response and calls the **serve()** polymorphed function. The function returns a Boost::status that we can implement in future iterations as an error check.



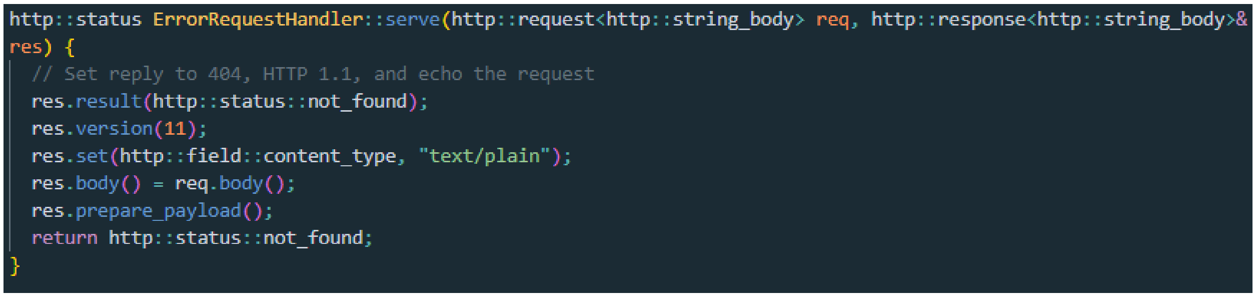
```
http::status RequestHandler::handle_request(http::request<http::string_body> req,
http::response<http::string_body>& res) {
  http::status return_status = serve(req, res);
  return return_status;
}
```

Echo Handler



```
http::status EchoRequestHandler::serve(http::request<http::string_body> req, http::response<http::string_body>&
res) {
  // Set response to 200 OK, HTTP 1.1, and echo the request
  res.result(http::status::ok);
  res.version(11);
  res.set(http::field::content_type, "text/plain");
  res.body() = req.body();
  res.prepare_payload();
  return http::status::ok;
}
```

Error Handler

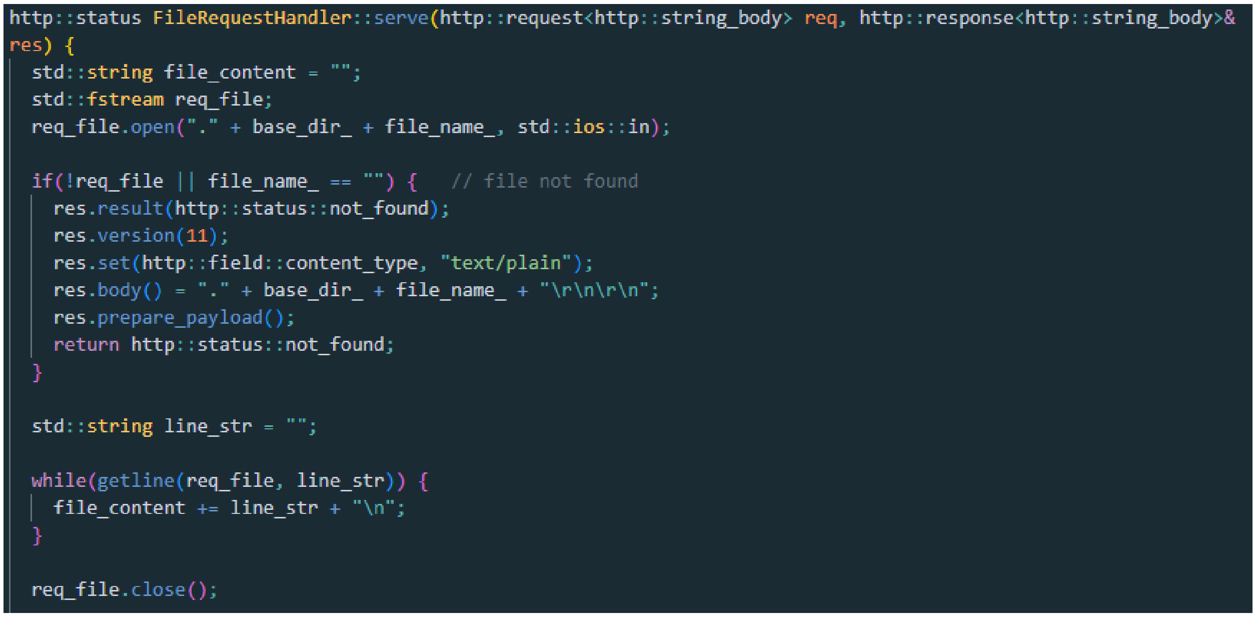


```
http::status ErrorRequestHandler::serve(http::request<http::string_body> req, http::response<http::string_body>&
res) {
  // Set reply to 404, HTTP 1.1, and echo the request
  res.result(http::status::not_found);
  res.version(11);
  res.set(http::field::content_type, "text/plain");
  res.body() = req.body();
  res.prepare_payload();
  return http::status::not_found;
}
```

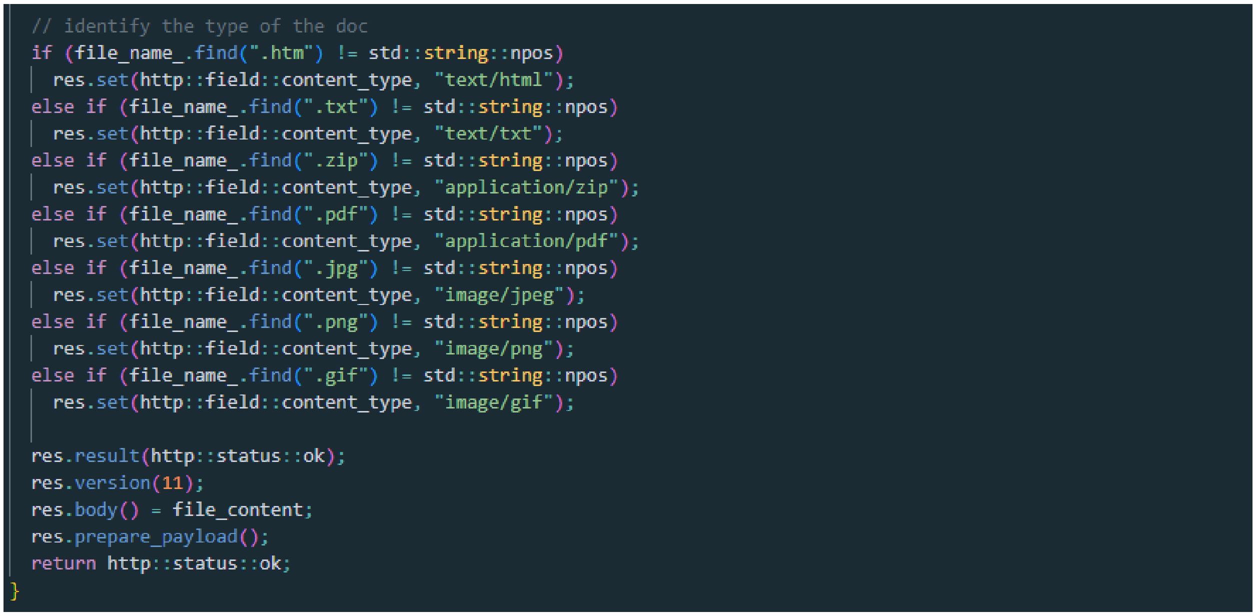
Both the Echo and Error Handlers are simple to implement with the only difference being the result() set to different statuses. Both responses return a **Content-Type** of **text/plain** along with the request.

File Handler

The File Handler is more complex as we need to read in the requested file and set the body of the response as the file contents. There is also a check at the beginning of the function to ensure that the request file exists.



```
http::status FileRequestHandler::serve(http::request<http::string_body> req, http::response<http::string_body>&
res) {
  std::string file_content = "";
  std::fstream req_file;
  req_file.open("." + base_dir_ + file_name_, std::ios::in);

  if(!req_file || file_name_ == "") {   // file not found
    res.result(http::status::not_found);
    res.version(11);
    res.set(http::field::content_type, "text/plain");
    res.body() = "." + base_dir_ + file_name_ + "\r\n\r\n";
    res.prepare_payload();
    return http::status::not_found;
  }

  std::string line_str = "";

  while(getline(req_file, line_str)) {
    file_content += line_str + "\n";
  }

  req_file.close();
```

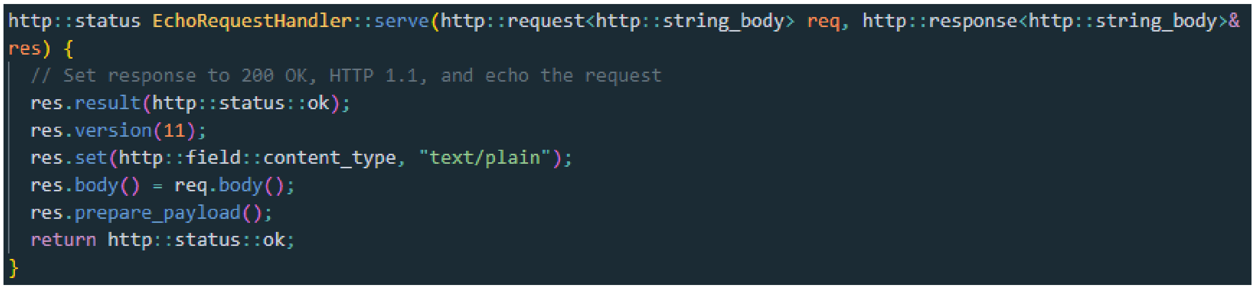


```
  // identify the type of the doc
  if (file_name_.find(".htm") != std::string::npos)
    res.set(http::field::content_type, "text/html");
  else if (file_name_.find(".txt") != std::string::npos)
    res.set(http::field::content_type, "text/txt");
  else if (file_name_.find(".zip") != std::string::npos)
    res.set(http::field::content_type, "application/zip");
  else if (file_name_.find(".pdf") != std::string::npos)
    res.set(http::field::content_type, "application/pdf");
  else if (file_name_.find(".jpg") != std::string::npos)
    res.set(http::field::content_type, "image/jpeg");
  else if (file_name_.find(".png") != std::string::npos)
    res.set(http::field::content_type, "image/png");
  else if (file_name_.find(".gif") != std::string::npos)
    res.set(http::field::content_type, "image/gif");

  res.result(http::status::ok);
  res.version(11);
  res.body() = file_content;
  res.prepare_payload();
  return http::status::ok;
}
```

3 How to add a Request Handler

To add a new request handler, simply create a new header and source code file. Take the existing serve function from the parent interface **request_handler.h** and determine what type of requests it will handle.

Using the Echo Request Handler source code as a template, set the response attributes with appropriate values.



```
http::status EchoRequestHandler::serve(http::request<http::string_body> req, http::response<http::string_body>&
res) {
  // Set response to 200 OK, HTTP 1.1, and echo the request
  res.result(http::status::ok);
  res.version(11);
  res.set(http::field::content_type, "text/plain");
  res.body() = req.body();
  res.prepare_payload();
  return http::status::ok;
}
```