Kapitel 1

RESTinio

RESTinio ist eine header-only Bibliothek, mit dem man REST - APIs schnell und einfach entwickeln kann und ebenso um Requests asynchron verarbeiten zu können.

Um RESTinio verwenden zu können, muss man neben RESTinio auch die Bibliotheken asio, http_parser und fmt inkludieren, da RESTinio diese verwendet.

1.1 Express

Mithilfe dieser Bibliothek kann man Server wie Node.js Express Server entwickeln.

1.1.1 main

Um einen Express Server zu starten, muss die main Funktion wie folgt aussehen:

```
1 #include "restinio/all.hpp"
2
3 int main() {
4     using router_t = restinio::router::express_router_t<>;
5     try {
6         using traits_t =
7             restinio::traits_t<
8                 asio_timer_manager_t,
9                 single_threaded_ostream_logger_t,
10                router_t >;
11
12        restinio::run(
13            restinio::on_this_thread<traits_t>()
14            .port(8080)
15            .address("localhost")
16            .request_handler(server_handler()));
17    } catch( const std::exception & ex ) {
18        cerr << "Error: " << ex.what() << endl;
19        return 1;
20    }
21 }
```

Um RESTinio überhaupt verwenden zu können, muss man zuerst einmal ¨ restinio/all.hpp ¨ inkludieren. In main muss die Funktion **restinio::run** aufgerufen werden,

welche die Funktion **restinio::on_this_thread** erwartet. Diese Funktion wiederum erwartet zumindest 3 Parameter: den Port (8080), die Adresse (localhost) und den Request Handler, welcher alle eingehenden Request verarbeitet (server_handler()).

Bei Express Server benötigt man zusätzlich noch Traits, das wichtigste Trait wäre dabei **express_router_t**, mit diesem gibt man an, dass es sich um einen Express Router handelt.

1.1.2 server_handler()

Damit der Server weiß, welche Request man wie verarbeiten muss, definiert man einen Request Handler, dieser könnte wie folgt aussehen:

```
1 #include "restinio/all.hpp"
2
3 auto server_handler() {
4     auto router = std::make_unique< router_t >();
5
6     //GET request for route /
7     router->http_get("/", [](auto req, auto) {
8         return req->create_response()
9             .set_body("Hello World")
10            .done();
11    });
12
13    //will be called, if route doesn't match exisitng ones
14    router->non_matched_request_handler(
15        []( auto req ){
16            return
17                req->create_response(status_not_found())
18                    .append_header_date_field()
19                    .connection_close()
20                    .done();
21        });
22    return router;
23 }
```

In dieser Funktion muss man eine Variable **router** definieren, mit dieser kann man weitere Handler definieren. Diese Handler bearbeiten den jeweiligen Request und schicken einen entsprechenden Response zurück.

In diesem Beispiel wurde nur ein Handler für einen GET Request auf / definiert. Wenn dieser aufgerufen wird, dann wird ein Response mithilfe der Funktion **create_response()** erstellt, der Body dieses Response wird auf ¨ Hello World ¨ gesetzt und anschließend wird dieser an den Client gesendet.

Falls aber ein Request auf eine undefinierte Route erstellt wird (z.B. GET /material), werden diese in der Funktion **non_matched_request_handler** bearbeitet. In dem Beispiel wird einfach nur der Status-Code ¨ 404 Not Found ¨ gesetzt und die Verbindung geschlossen.

1.1.3 Header und Query Parameter

Bei Requests gibt es 2 Arten von Parametern: Header und Query Parameter.

Header Parameter werden im Request Header mitgeschickt. Mit den Methoden get_field(string headername) und has_field(string headername) kann man auf seinen Wert zugreifen und man kann überprüfen, ob dieser existiert.
Query Parameter werden in der URL mitgeschickt. Diese werden mit einem Doppelpunkt nach einem / gekennzeichnet und man kann auf sie mit dem Funktionsparameter params[¨ parametername¨] zugreifen.

Im folgenden Beispiel werden beide Arten von Parametern verwendet. Wenn der User über einen Authorization Header verfügt, sendet der Server einen Request mit seinem Wert und mit dem Query Parameter Wert zurück, wenn nicht, sendet der Server den Status-Code ¨ 401 Unauthorized¨ zurück:

```
1 router->http_get("/single/:name", [](auto req, auto params ){
2     if (req->header().has_field("Authorization")) {
3         string auth{req->header().get_field("Authorization")};
4 
5         return init_resp(req->create_response())
6                .set_body(auth + "," + params["name"])
7                .done();
8     } else {
9          return req->create_response(status_unauthorized()).done();
10    }
11 });
```

1.2 Senden von Dateien

Mit RESTinio kann man ebenso Dateien vom Client zum Server senden und umgekehrt.

Um Inhalte von Dateien **vom Server zum Client** zu senden, muss man einfach mithilfe der Funktion ¨ sendfile¨ den Body der Funktion setzen. Die Funktion sendfile erwartet sich dabei den Pfad zur Datei, die man senden will.

```
1 #include "restinio/all.hpp"
2 
3 router->http_get("/materialien", [](auto req, auto) {
4     return req->create_response()
5         .set_body(restinio::sendfile("materialien.csv"))
6         .done();
7 });
```

Um Inhalte von Dateien **vom Client zum Server** zu senden, verwendet man die Funktion ¨ enumarate_parts_with_files ¨ . Diese Funktion liefert ein ¨ expected_t<std::size_t, enumeration_error_t>¨ zurück, bei erfolgreichem Verarbeiten (keine Fehler treten auf) wird also die Anzahl der Body Parts zurückgeliefert, bei nicht erfolgreichem Verarbeiten wird der Wert von ¨ enumeration_error_t¨ zurückgeliefert.

Im folgenden Beispiel wird die Datei in Zeilen unterteilt und alle Zeilen werden in der Funktion ¨ doit¨ verarbeitet. Den Inhalt der Datei bekommt man mit "part.body", der Inhalt wurde mithilfe der ¨ split¨ Funktion von pystring aufgeteilt (pystring ist nicht in RESTinio enthalten).

```
#include "restinio/all.hpp"
#include "restinio/helpers/multipart_body.hpp"
#include "restinio/helpers/file_upload.hpp"

router->http_get("/", [](auto req, auto) {
    using namespace restinio::file_upload;

    const auto result = enumerate_parts_with_files( *req,
        [](part_description_t part) {
        //split file into lines
        vector<string> lines;
        pystring::split(string(part.body), lines, "\n");

        //do something with each line
        for (string line : lines) {
            doit(line);
        }
        return handling_result_t::continue_enumeration;
    });

    if(result) {
        ... // Producing positive response.
    } else {
        ... // Producing negative response.
    }
});
```

1.3 TLS

RESTinio unterstützt ebenso HTTPS mithilfe von asio (basierend auf OpenSSL). Um einen Server als HTTPS Server anzugeben, muss man "restinio::tls_traits_töder "restinio::single_thread_tls_traits_t"verwenden. Ebenso muss man den TLS Context setzen.

```
#include "restinio/all.hpp"
#include "restinio/tls.hpp"

int main() {
    try {
        //defining the traits
        using traits_t =
            restinio::tls_traits_t<
                asio_timer_manager_t,
                single_threaded_ostream_logger_t,
                router_t >;

        //create and set options of tls_context
        asio::ssl::context tls_context{ asio::ssl::context::sslv23 };
        tls_context.set_options(
        asio::ssl::context::default_workarounds |
        asio::ssl::context::no_sslv2 |
        asio::ssl::context::single_dh_use );

        string certs_dir{"../src/server/certs"};

        //use your own certificates for the server
        tls_context.use_certificate_chain_file(certs_dir + "/server.pem");
        tls_context.use_private_key_file(
            certs_dir + "/key.pem",
            asio::ssl::context::pem);
        tls_context.use_tmp_dh_file(certs_dir + "/dh2048.pem");

        //run the server with your tls_context
        run(on_this_thread<traits_t>()
            .port(1234).address("localhost")
            .request_handler(server_handler())
            .tls_context(std::move(tls_context)));
    } catch(const std::exception & ex) {
        cerr << "Error: " << ex.what() << endl;
        return 1;
    }
}
```