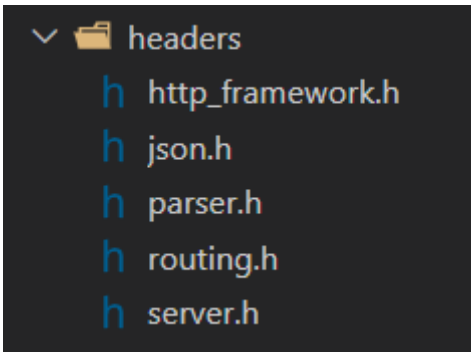
Perancangan Framework Backend Server Menggunakan Socket pada Bahasa C

The sister.js in question:

Disusun sebagai tugas Seleksi Laboratorium Sistem Terdistribusi 2024

- pol

A. Struktur Framework



Framework backend yang dirancang terdiri atas lima file header seperti yang terlihat di gambar.

http_framework.h

File header ini menyimpan seluruh struktur data penting yang akan digunakan untuk implementasi framework oleh penulis framework maupun server untuk penulis backend. Struktur data paling penting pada framework ini adalah struktur **HTTPRequest dan HTTPResponse.**

```c
// Structure to hold HTTP request data
typedef struct {
    // First line
    HTTPMethod method;
    char path[256];
    char version[16];

    // Headers
    HTTPHeader headers[MAX_HEADERS];
    int header_count;

    // Query parameters
    HTTPQueryParameter query_params[MAX_QUERY_PARAMS];
    int query_param_count;

    // Body
    char body[BUFFER_SIZE];
} HTTPRequest;
```

Struktur HTTPRequest menyimpan seluruh informasi yang diberikan client ketika melakukan request ke server. Sebuah HTTP request umumnya dibagi menjadi tiga bagian, yaitu baris pertama, headers, body.

Baris pertama mengandung informasi metode request (GET, POST, PUT, DELETE), path request (/home, /about, dll.), serta versi HTTP yang dipakai. Bagian headers menyimpan metadata terkait request yang dikirim, seperti jenis body, host, waktu kirim, dan lain-lain. Bagian terakhir adalah body sendiri yang tidak selalu ada pada seluruh request (umumnya hanya pada POST dan PUT), namun akan diisi oleh

informasi penting yang ingin diunggah oleh client. Selain ketiga bagian tersebut, sebuah HTTP request juga dapat memiliki parameter yang bersifat opsional.

```c
// Structure to hold HTTP response data
typedef struct {
    // First line
    int status_code;
    char status_message[64];

    // Headers
    HTTPHeader headers[MAX_HEADERS];
    int header_count;

    // Body
    char body[BUFFER_SIZE];
} HTTPResponse;
```

Untuk struktur HTTPResponse, yang menyimpan informasi terkait respon server kepada client, menyimpan data-data yang tidak jauh beda dengan HTTPRequest. Perbedaan utama ada di baris pertama, yang sekarang menyimpan status code (200 OK, 404 Not Found, dst.), serta pesan singkat terkait status respons. HTTP response juga tidak memiliki parameter.

parser.h

Header file ini berperan sebagai pengatur manipulasi string program, dan menyimpan dua prosedur penting, yaitu **parse_request** untuk melakukan parsing terhadap string HTTP Request yang dikirim client, serta **build_response** untuk membangun string yang akan dikirim ke client sebagai HTTP Response.

routing.h

File ini berperan dalam pelayanan HTTPRequest dengan menyalurkan request yang dikirim client ke sebuah **backend route** yang sesuai. Backend route ini dapat dianggap sebagai jembatan antar HTTP Request dan fungsi layanan yang bersesuaian pada server.

```c
// Structure to hold a route
typedef struct {
    // Route path
    char path[256];

    // Route HTTP method
    HTTPMethod method;

    // Route handler function
    void (*handler)(HTTPRequest *request, HTTPResponse *response);
} Route;

// Route table
Route routes[MAX_ROUTES];
int route_count = 0;
```

Sebuah rute dapat dianggap sebagai kombinasi method dan path yang di-map ke sebuah prosedur. Server akan menyimpan sebuah array of routes dan untuk setiap HTTP Request yang masuk, akan dilakukan iterasi pada array tersebut untuk mencari prosedur yang tepat.

json.h

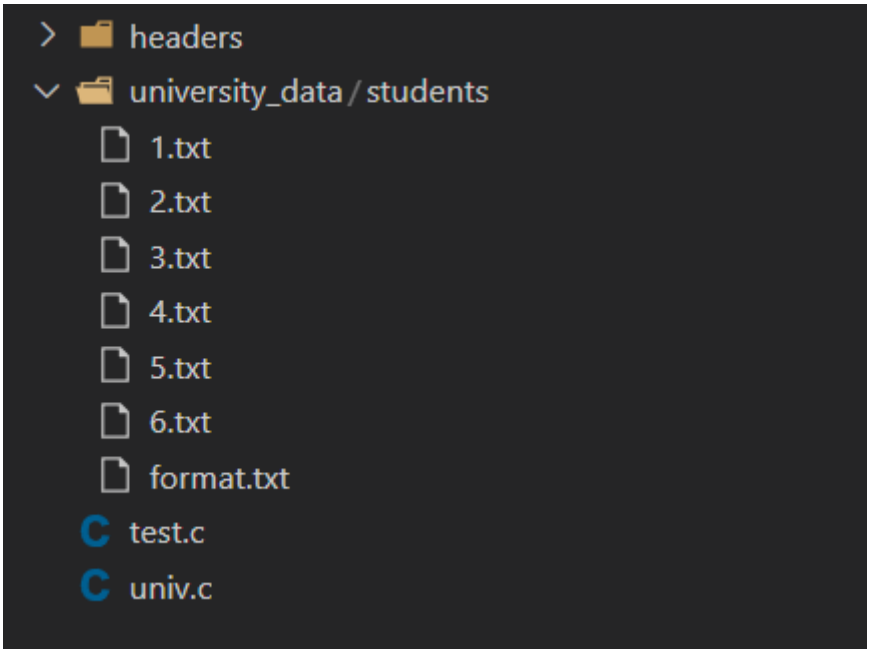
```c
// Structure for JSON attributes
typedef struct {
    char key[256];
    char value[4096];
} JSONAttribute;

// Structure for JSON object
typedef struct {
    JSONAttribute attr[MAX_JSON_ATTR];
    int attr_count;
} JSONObject;
```

File header ini menyimpan struktur data dan prosedur untuk memudahkan operasi dengan tipe data JSON. Sebuah objek JSON pada framework ini hanyalah sebuah struktur yang menyimpan *an array of key-value pairs*, atau pada framework ini disebut attribute. Key-value pada atribut didefinisikan sebagai string, sehingga sebuah atribut JSON akan selalu diasumsikan memiliki key dan value bertipe string.

Selain struktur data, juga terdapat prosedur **json_stringify** dan **json_parse** untuk melakukan string building dan parsing terhadap JSON. Juga terdapat prosedur **json_add_attr** untuk menambahkan atribut pada sebuah objek JSON.

server.h
File header ini menyimpan logika untuk handling penerimaan request dan pengiriman response pada server, sehingga implementasi socket untuk komunikasi client-server akan sepenuhnya berada di sini. Penjelasan terkait implementasi socket dan komunikasi akan dibahas selanjutnya.

B. Fitur-Fitur yang Diimplementasikan



Untuk pengetesan fitur-fitur framework, terutama handling method-method GET / POST / PUT / DELETE, saya menggunakan sebuah contoh implementasi backend berupa server universitas yang menyimpan data mahasiswa. Server ini dapat digunakan client untuk membaca atau memodifikasi data mahasiswa menggunakan HTTP request.

Backend servernya adalah **univ.c** yang dapat di-compile dengan **gcc -o univ univ.c**. Lalu server cukup dijalankan dengan **./univ**. Server ini akan mengimplementasikan framework backend yang sudah dibuat pada server **headers**. Data universitas disimpan di **university_data**, pada contoh ini hanya ada data **students** yang berisi data-data mahasiswa yang masing-masing terpisah pada file sendiri dan dinamakan berdasarkan **student_id**.

Implementasi Socket

```
// Main server code
int main() {
    add_route(GET, "/students", get_students_handler);
    add_route(POST, "/students", post_students_handler);
    add_route(PUT, "/students", put_students_handler);
    add_route(DELETE, "/students", delete_students_handler);

    start_server(8080);
    return 0;
}
```

Komunikasi antar client-server akan dilakukan melalui **socket** yang di-setup pada server. Untuk menjalankan server menggunakan framework, cukup panggil **start_server** dengan parameter **nomor port yang diinginkan**.

```
server_socket = socket(AF_INET, SOCK_STREAM, 0);
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(port);
server_addr.sin_addr.s_addr = INADDR_ANY;
```

Bagian ini melakukan setup pada **server socket** dan **server address**, menetapkan server untuk menggunakan **TCP (SOCK_STREAM)**, **IPv4 (AF_INET)**, serta menetapkan port untuk server mendengarkan koneksi pada **htons(port)**.

```
bind(server_socket, (struct sockaddr*)&server_addr, sizeof(server_addr));
listen(server_socket, 10);
```

Bagian ini akan mengikat socket dengan alamat port **(bind)**, serta menyuruh server untuk mulai mendengarkan socket **(listen)** dengan maksimum 10 client pada satu waktu.

```
while (1) {
    socklen_t addr_len = sizeof(client_addr);
    client_socket = accept(server_socket, (struct sockaddr*)&client_addr, &addr_len);
```

Server akan memulai looping untuk mendengarkan request dari client. Fungsi **accept** akan melakukan blocking sampai didapatkan koneksi dari sebuah client.

```
ssize_t bytes_received = recv(client_socket, buffer, sizeof(buffer) - 1, 0);
if (bytes_received < 0) {
    perror("recv failed");
    close(client_socket);
    continue;
}
// Null-terminate the buffer to prevent garbage values
buffer[bytes_received] = '\0';
```

Ketika didapatkan koneksi, maka server akan menerima requestnya pada **client_socket** dan memasukannya ke **buffer**.

```
// Parse and process request
parse_request(buffer, &request);
memset(&response, 0, sizeof(response));
route_dispatcher(&request, &response);
```

Dari request yang didapat pada buffer, server akan melakukan **parsing** terhadap request tersebut menjadi tipe data **HTTPRequest**. Selanjutnya, server akan memanggil **route_dispatcher** untuk mencari handler yang sesuai untuk request tersebut, memanggilnya, dan memasukkan response server ke buffer **response**.

```
// Build and send response
char response_buffer[BUFFER_SIZE];
build_response(&response, response_buffer);
printf("Sending HTTP response:\n%s\n", response_buffer);
send(client_socket, response_buffer, strlen(response_buffer), 0);

// Close the client socket
close(client_socket);
```

Lalu server akan membangun **string response** dan mengirimnya ke client melalui **client_socket**. Terakhir, server akan menutup socket client dengan **close(client_socket)** untuk menandakan berakhirnya komunikasi, dan server dapat melanjutkan ke mendengarkan request dari client.

Routing Request

```
// Function to find and dispatch the appropriate route handler
void route_dispatcher(HTTPRequest *request, HTTPResponse *response) {
    // Error 405: Method Not Allowed
    if (request->method == UNKNOWN) {···

    // Find the route handler
    for (int i = 0; i < route_count; i++) {
        if (strcmp(request->path, routes[i].path) == 0 && request->method == routes[i].method) {
            routes[i].handler(request, response);
            return;
        }
    }
```

Fitur ini diimplementasikan menggunakan **routes table** dan **prosedur route_dispatcher** pada file **routing.h**. Jika diberikan sebuah HTTP Request dengan method dan path tertentu, prosedur route_dispatcher akan melakukan iterasi seluruh route pada route table hingga ditemukan route yang sesuai, yang kemudian request dan response akan dioper ke handler function pada route tersebut.

```
// Main server code
int main() {
    add_route(GET, "/students", get_students_handler);
    add_route(POST, "/students", post_students_handler);
    add_route(PUT, "/students", put_students_handler);
    add_route(DELETE, "/students", delete_students_handler);
```

Cara menggunakan routing cukup dengan memanggil prosedur **add_route** yang akan menambahkan kombinasi method, path, dan handler ke route table. Setiap HTTP Request yang masuk ke server akan secara otomatis dikirimkan ke route_dispatcher untuk pencarian request handler yang tepat.

Method GET

```
// Main server code
int main() {
    add_route(GET, "/students", get_students_handler);
```

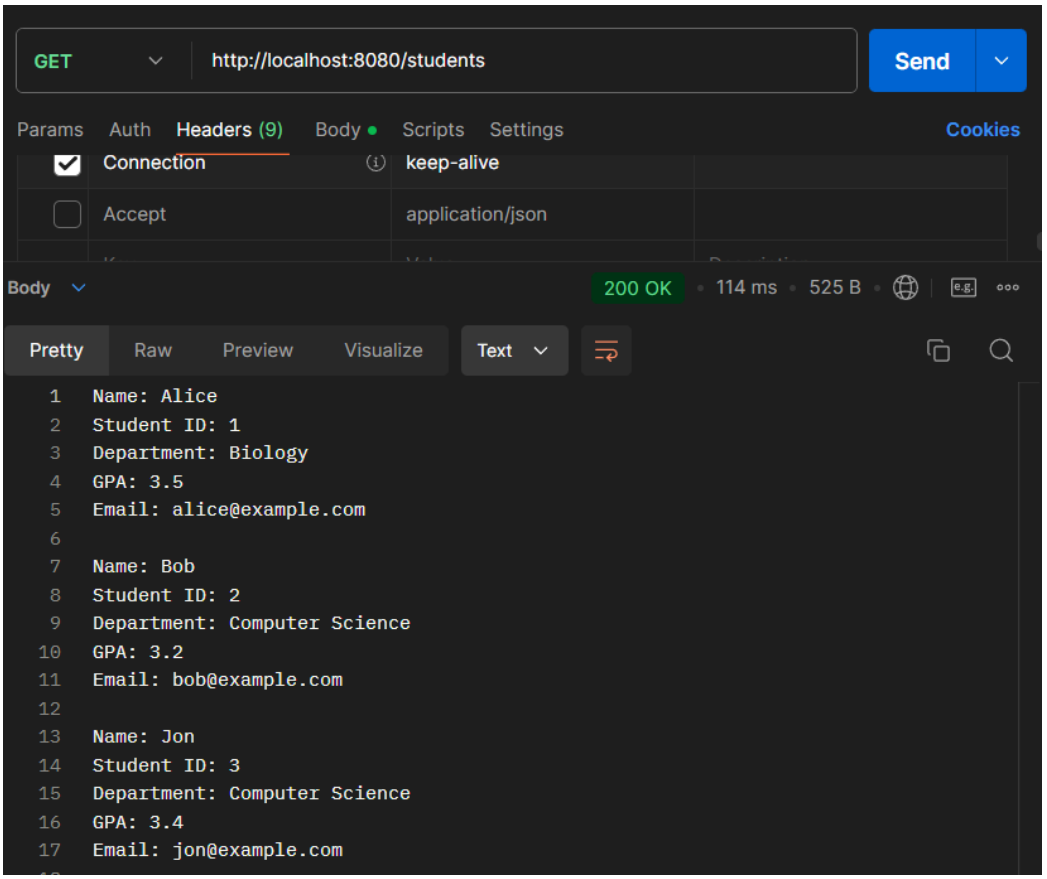
Pada server universitas, request **GET /students** dapat digunakan untuk mendapatkan data mahasiswa. Fitur ini diimplementasikan pada server cukup dengan memanggil fungsi **add_route** dan membuat sebuah fungsi handler **get_students_handler**.

```c
void get_students_handler(HTTPRequest* req, HTTPResponse* res) {
    res->status_code = 200;
    strcpy(res->status_message, "OK");

    // Read student data
    Student students[MAX_STUDENTS]; memset(students, 0, sizeof(students));
    int student_count = 0;

    read_students("./university_data/students", students, &student_count, req);
```

Implementasi untuk pembacaan data mahasiswa cukup sederhana, handler function untuk GET /students akan memanggil fungsi lain yaitu **read_students** yang akan membaca keseluruhan data mahasiswa dan memasukkannya pada array **students**. Fungsi read_students lalu akan memanggil **read_student_file** yang akan membuka dan membaca masing-masing file mahasiswa.

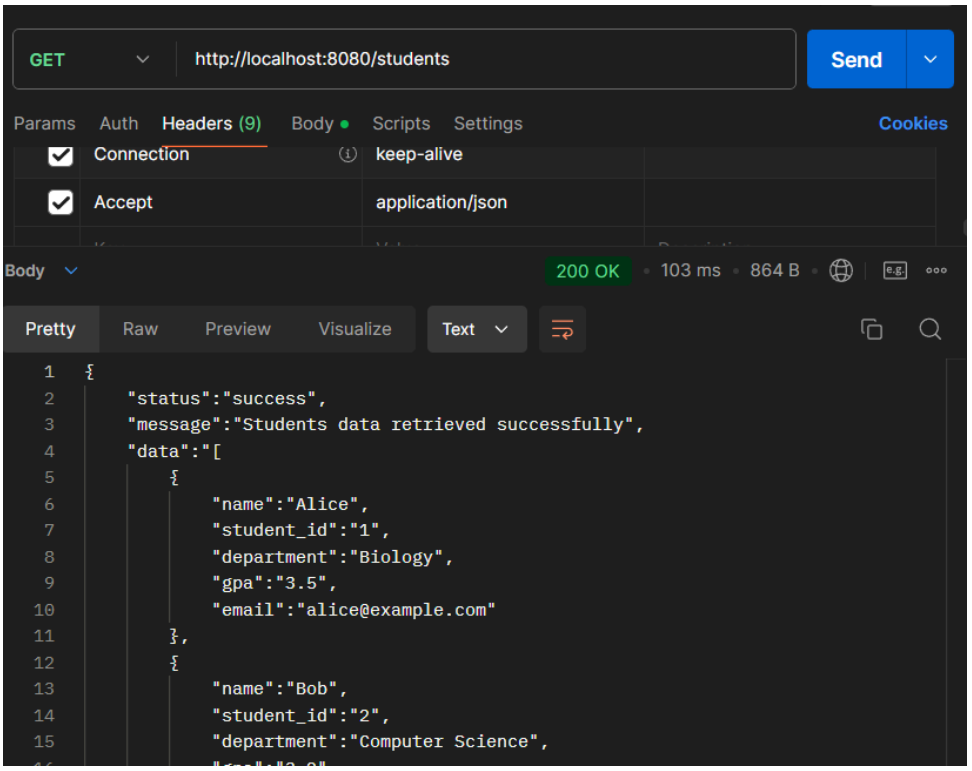
```c
// Get students
int read_student_file(const char* filepath, Student* student, HTTPRequest* req) {···

void read_students(const char* path, Student* students, int* student_count, HTTPRequest* req) {··
```

Setelah didapatkan keseluruhan data mahasiswa, fungsi handler cukup membangun sebuah string response yang akan dikirim ke client berisi data-data mahasiswa tersebut.

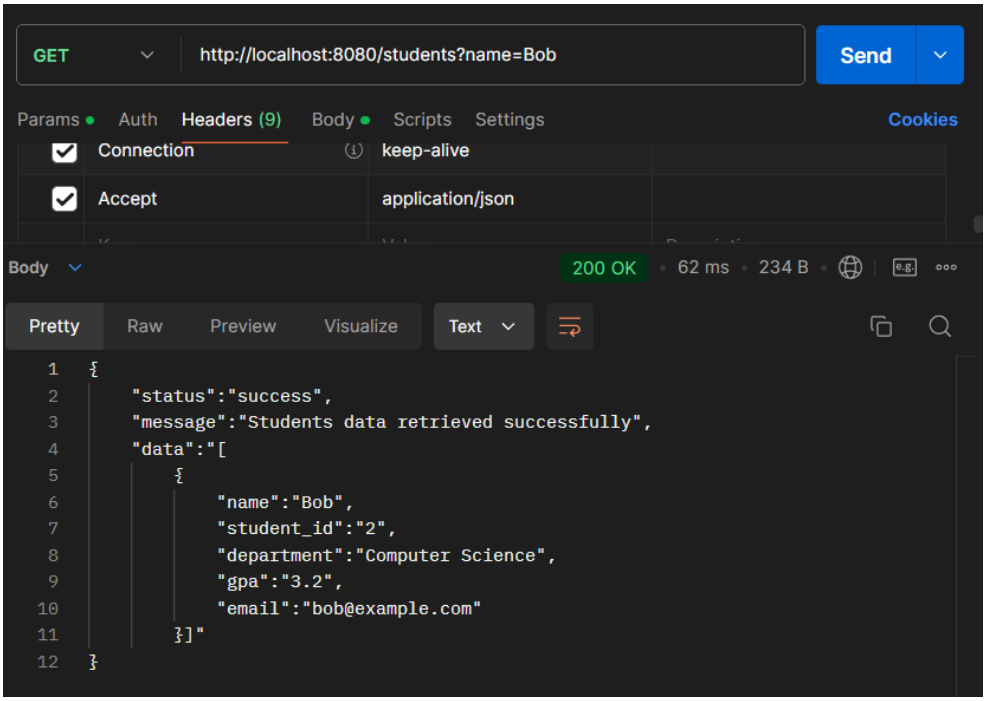


Pemakaian metode GET cukup dengan mengirim HTTP GET ke salah satu rute yang sudah didefinisikan di server, pada kasus ini **GET /students**. Server akan mengirim

HTTP Response berupa seluruh mahasiswa yang terdata pada server. Header **Accept** pada request dapat diganti dari */* menjadi **application/json** untuk mendapatkan response dalam bentuk JSON.



Implementasi GET pada server ini juga dapat menerima query parameter, contohnya **GET /students?name=Bob**, yang server hanya akan mengembalikan seluruh data mahasiswa bernama 'Bob'. Penambahan query parameter pada request cukup dengan menambahkan **?key1=value1&key2=value2&key3=value3&...** pada path request.



```c
// Parse query parameters
query++;
int i = 0;
while (*query != '\0') {
    int len = 0;
    while (*query != '=') {
        request->query_params[i].key[len] = *query;
        query++;
        len++;
    }

    query++;
    len = 0;
    while (*query != '&' && *query != '\0') {
        request->query_params[i].value[len] = *query;
        query++;
        len++;
    }
    i++;
    if (*query == '&') query++;
}
```

Implementasi fitur untuk parsing query parameter cukup sederhana, dengan mengambil bagian path setelah tanda '?', lalu melakukan parsing per karakter. Sebuah key akan di-parse sampai karakter '=', lalu parser akan berpindah ke parsing value, yang akan di-parse sampai ditemukan karakter '&' atau '\0'. Jika pada akhir value terdapat karakter '&', maka parsing dilanjutkan ke pasangan key-value selanjutnya.

Method POST

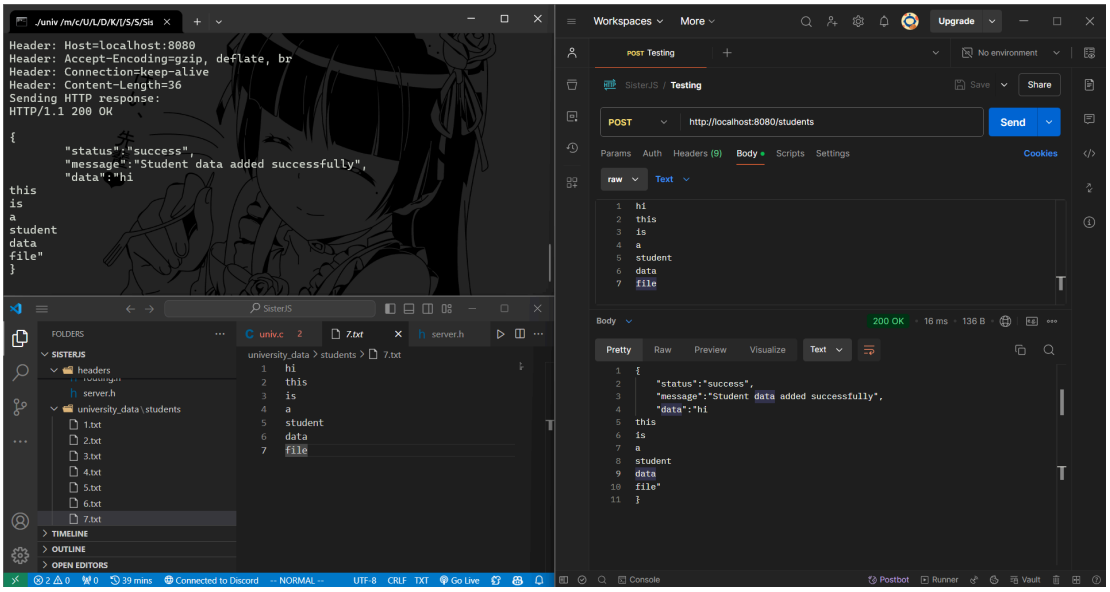
```c
// Main server code
int main() {
    add_route(GET, "/students", get_students_handler);
    add_route(POST, "/students", post_students_handler);
```

Request **POST /students** juga diimplementasikan pada server menggunakan fungsi add_route, dengan **post_students_handler** sebagai fungsi handler. Rute ini dapat digunakan client untuk menambah data mahasiswa baru ke server.

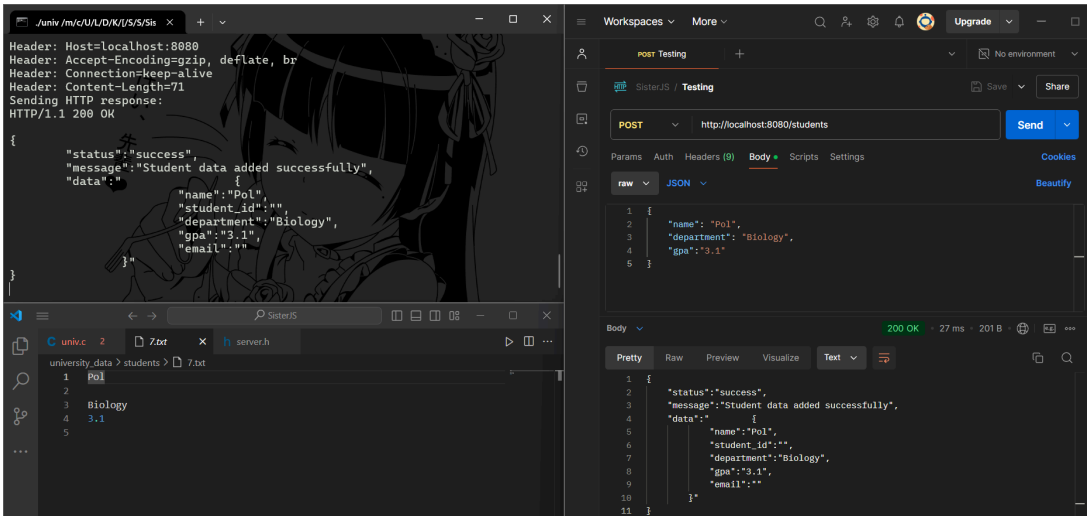
Client akan memasukkan data mahasiswa pada bagian body POST request. Contoh data mahasiswa yang digunakan pada server adalah: **name, student_id, department, gpa,** dan **email**. Ketika memproses request, server pertama akan mengecek apakah parameter student_id diberikan oleh client. Jika tidak diberikan, maka server akan menetapkan **student_count + 1** sebagai id mahasiswa tersebut.

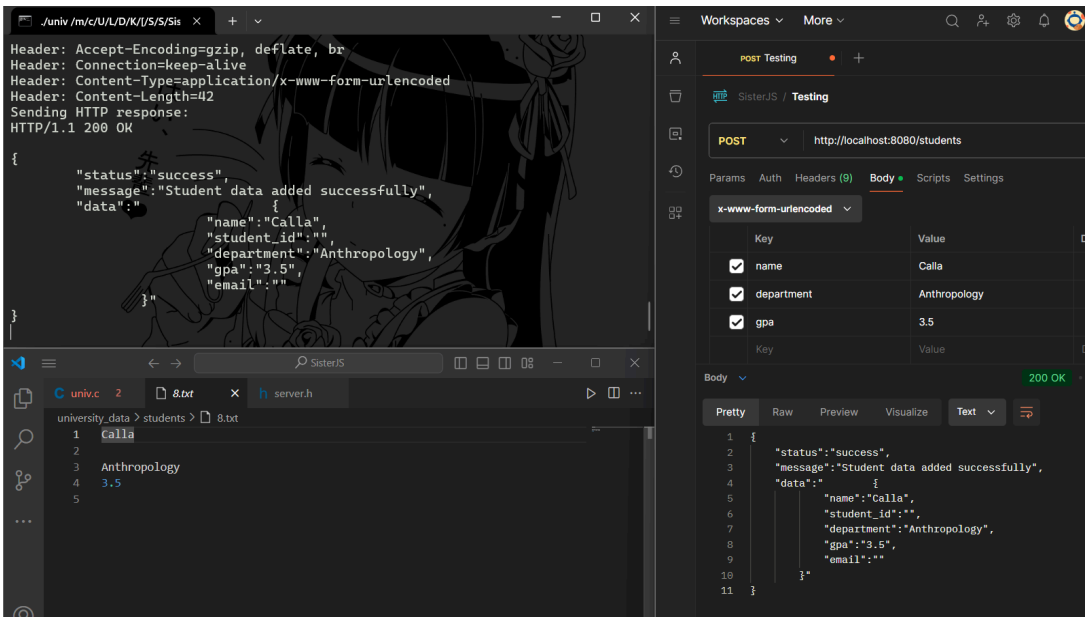
Pada kedua kondisi, server akan menyimpan data mahasiswa pada file [student_id].txt. Jika file tersebut sudah ada di server, alias student_id sudah terpakai, maka server akan mengembalikan error **409 Conflict: Student ID already exists** dan data mahasiswa tidak disimpan.

Format data yang diberikan client dapat berupa salah satu dari tiga tipe: **text/plain, application/json,** atau **application/x-www-form-urlencoded**.



Jika client menggunakan format **text/plain**, maka body request tidak akan di-parsing dan request body akan ditulis langsung ke file saja. Untuk format ini client tidak dapat memberikan student_id ke server, sehingga server akan otomatis menggunakan id = student_count + 1.





Format **application/json** dengan **application/x-www-form-urlencoded** tidak jauh berbeda, karena keduanya merupakan format key-value pair. Perbedaannya hanya pada metode parsing-nya saja. Untuk **format application/json** akan menggunakan json_parser yang akan dibahas pada bagian **Parser JSON**.

```c
void query_parse(JSONObject *json, char *buffer) {
    while (*buffer != '\0') {
        int len = 0;
        while (*buffer != '=') {
            json->attr[json->attr_count].key[len] = *buffer;
            buffer++;
            len++;
        }

        buffer++;
        len = 0;
        while (*buffer != '&' && *buffer != '\0' && *buffer != '\r' && *buffer != '\n') {
            json->attr[json->attr_count].value[len] = *buffer;
            buffer++;
            len++;
        }

        json->attr_count++;
        if (*buffer == '&') buffer++;
    }
}
```

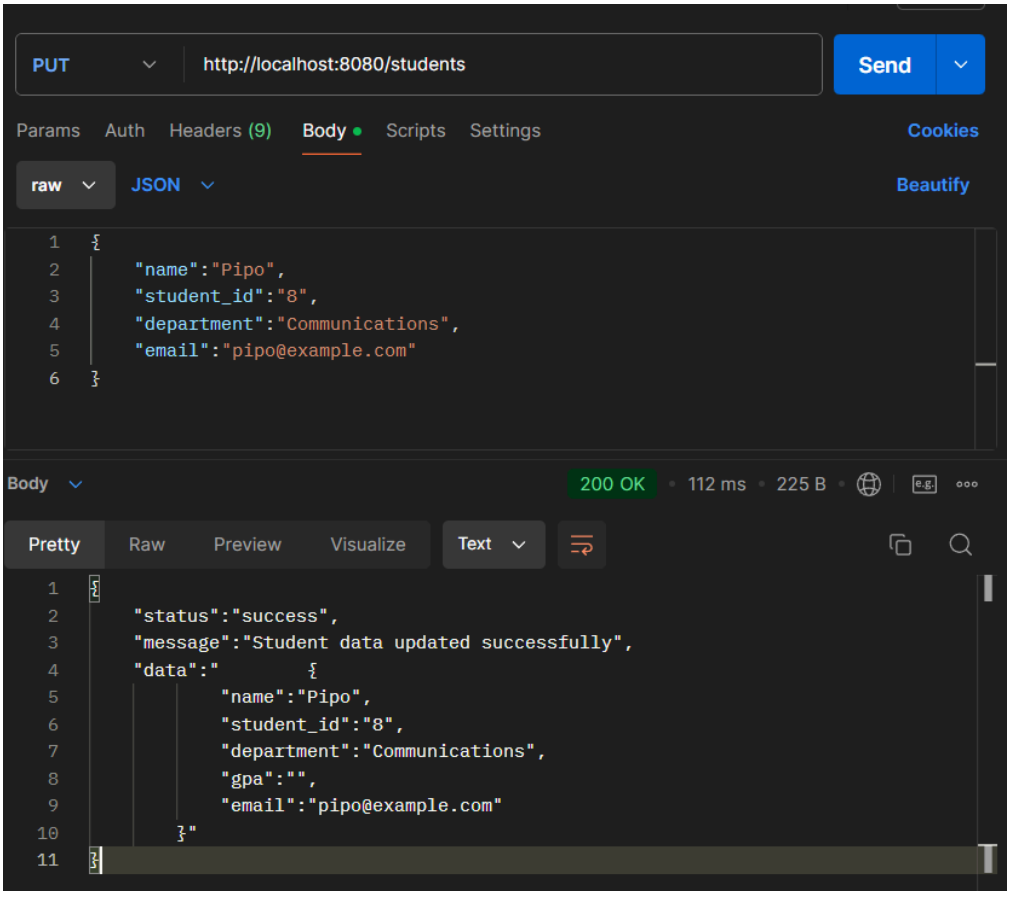
Parsing untuk **format application/x-www-form-urlencoded** tidak jauh berbeda dari metode parsing untuk query parameter, karena format key-value pada tipe body ini juga menggunakan format **key1=value1&key2=value2&...**, sehingga parser untuk format ini hampir sama persis secara kode dengan parser query parameter.

Kedua format ini dapat melakukan parsing untuk parameter berikut: **name, student_id, department, gpa,** dan **email**. Karena dapat memberikan student_id secara langsung, maka client dapat menentukan sendiri student_id mahasiswa, namun jika tidak diberikan maka akan tetap digunakan id = student_count + 1.

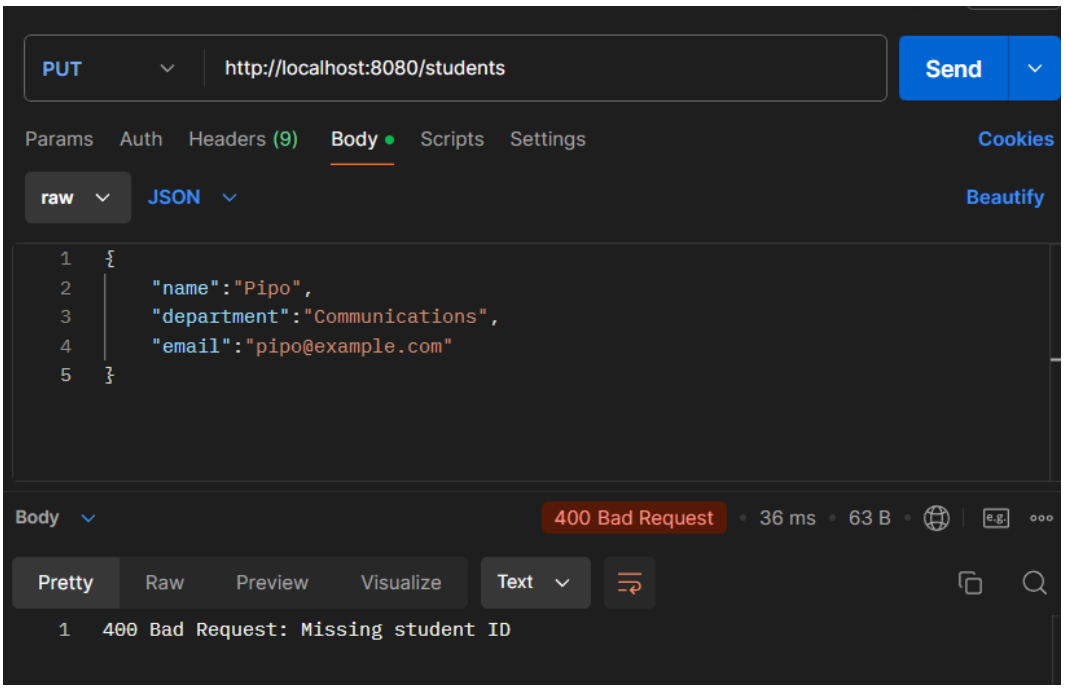
Method PUT

```
// Main server code
int main() {
    add_route(GET, "/students", get_students_handler);
    add_route(POST, "/students", post_students_handler);
    add_route(PUT, "/students", put_students_handler);
```

Implementasi **PUT /students** sama seperti metode lainnya, dengan menambahkan rute menggunakan fungsi add_route. Secara implementasi, metode PUT hampir sama persis dengan metode POST, hanya saja metode ini digunakan untuk modifikasi data, bukan untuk menambahkan data.



Perbedaan utama pada implementasi PUT /students adalah pada pemberian student_id oleh client. Karena metode ini bersifat memodifikasi data yang spesifik, maka student_id harus diberikan pada request. Untuk format **text/plain**, student_id diberikan sebagai query parameter (PUT /students?id=...). Untuk format **application/json** dan **application/x-www-form-urlencoded**, student_id tetap diberikan sebagai key, namun untuk metode ini bersifat wajib.



Jika request PUT yang dikirim tidak memiliki student_id, maka akan diberikan error **400 Bad Request: Missing student ID**, dan jika data student_id tidak ada pada server, akan diberikan error **404 Not Found: Student ID not found**.

Tidak ada perbedaan atas **parsing application/json dan parsing application/x-www-form-urlencoded** pada metode ini dibanding dengan pada metode POST.

Method DELETE

```
// Main server code
int main() {
    add_route(GET, "/students", get_students_handler);
    add_route(POST, "/students", post_students_handler);
    add_route(PUT, "/students", put_students_handler);
    add_route(DELETE, "/students", delete_students_handler);

    start_server(8080);
    return 0;
}
```

Untuk metode terakhir, **DELETE /students** juga diimplementasikan dengan fungsi add_route. Metode ini hampir sama dengan metode PUT /students, hanya saja melakukan penghapusan data secara total dibanding modifikasi saja. Perbedaan utama adalah metode ini tidak membutuhkan request body, sehingga tidak diperlukan parsing body.

Metode ini masih membutuhkan **student_id**, yang diberikan melalui query parameter. Sama seperti PUT /students, server akan mengembalikan **400 Bad**

Request: Missing student ID jika tidak diberikan student_id, dan **404 Not Found: Student ID not found** jika data tersebut tidak ditemukan.

Dynamic Routing

```
// Route table
Route routes[MAX_ROUTES];
int route_count = 0;

// Function to register a route
void add_route(HTTPMethod method, const char *path, void (*handler)(HTTPRequest *request, HTTPResponse *response)) {
    if (route_count < MAX_ROUTES) {
        // Request method
        routes[route_count].method = method;

        // Request path
        strcpy(routes[route_count].path, path);

        // Request handler
        routes[route_count].handler = handler;

        // Increment route count
        route_count++;
    }
}
```

Fungsi **add_route** yang sudah digunakan sebelumnya merupakan implementasi dari **dynamic routing**, yang diimplementasikan dengan menyimpan **list of routes**, dan untuk setiap HTTP request akan dicari route yang sesuai.

```
// Main server code
int main() {
    add_route(GET, "/students", get_students_handler);
    add_route(POST, "/students", post_students_handler);
    add_route(PUT, "/students", put_students_handler);
    add_route(DELETE, "/students", delete_students_handler);

    add_route(GET, "/professors", get_professors_handler);
    add_route(POST, "/professors", post_professors_handler);
    add_route(PUT, "/professors", put_professors_handler);
    add_route(DELETE, "/professors", delete_professors_handler);

    start_server(8080);
    return 0;
}
```

Penggunaan dynamic routing, sama seperti sebelumnya, cukup dengan menggunakan fungsi add_route tersebut. Contohnya jika ingin ditambahkan data dosen universitas pada server, cukup mendefinisikan handler-handlernya dan menambahkan route-routenya di server.

Parser JSON

```json
{
    "name":"Pipo",
    "student_id":"20",
    "department":"Communications",
    "email":"pipo@example.com"
}
```

```c
void json_parse(JSONObject *json, char *buffer) {
    while (*buffer != '\0') {
        // Increment until "
        while (*buffer != '\"') buffer++;
        buffer++;

        // Get key
        int len = 0;
        while (*buffer != '\"') {
            json->attr[json->attr_count].key[len] = *buffer;
            buffer++;
            len++;
        }
        buffer++;
        json->attr[json->attr_count].key[len] = '\0';

        // Increment until :
        while (*buffer != ':') buffer++;
        buffer++;

        // Increment until "
        while (*buffer != '\"') buffer++;
        buffer++;

        // Get value
        len = 0;
        while (*buffer != '\"') {
            json->attr[json->attr_count].value[len] = *buffer;
            buffer++;
            len++;
        }
        buffer++;
        json->attr[json->attr_count].value[len] = '\0';

        json->attr_count++;
        if (*buffer != ',') return;
    }
```

Parser JSON pada framework ini mengasumsikan bahwa value dari atribut selalu berupa tipe string. Cara kerja parser JSON memiliki logika yang sama dengan parser query parameter.

Pertama parser akan looping increment buffer sampai ditemukan karakter [“], yang menandakan key pertama. Seluruh karakter setelahnya akan ditambahkan ke key pertama, sampai ditemukan [“] selanjutnya, yang menandakan akhir dari parsing key pertama. Selanjutnya dimulai looping increment buffer lagi sampai ditemukan [“]

selanjutnya yang menandakan value pertama. Cara pengambilan value sama dengan pengambilan key, sampai ditemukan [“] pengakhir.

Setelah itu parser akan mengecek apakah karakter selanjutnya adalah [,] atau bukan. Jika iya, menandakan bahwa parser akan mengambil key-value selanjutnya. Jika tidak, maka parsing selesai.

Membuat Program di C
Program dibuat di C :D