Laboratorio Di Reti e Sistemi Distribuiti

Progetto di fine corso

Ilacqua Giovanni

Università degli Studi di Messina

6 luglio 2025

Introduzione al problema

Il progetto consiste nello sviluppo di un server HTTP concorrente, scritto in linguaggio C, conforme allo standard HTTP/1.1.

Il server deve gestire i seguenti tipi di richiesta: **GET**, **POST**, **PUT** e **DELETE**, e restituire risposte come: 200, 201, 204, 400.

L'obiettivo è creare un server HTTP minimale capace di inviare documenti HTML salvati localmente e registrare coppie chiave-valore sul server, gestendo più connessioni simultanee in modo concorrente.

Stato dell'Arte: HTTP

HTTP (Hypertext Transfer Protocol) è un protocollo del livello Applicazione ed è il protocollo che definisce il Web.

HTTP viene principalmente utilizzato per lo scambio di documenti ipertestuali in HTML. Tuttavia, le richieste e risposte definite da HTTP lo rendono versatile, ed è usato anche per altri scopi.

È basato su un'architettura client-server, dove il client (di solito un browser) invia richieste specificando metodo, risorsa, header e corpo. Il server risponde con uno stato, header e corpo.

HTTP è stateless, quindi non mantiene stato tra due richieste successive: per gestire una sessione servono meccanismi aggiuntivi.

HTTP: Trasporto e Sicurezza

HTTP utilizza TCP come protocollo di trasporto, che garantisce l'affidabilità delle comunicazioni, anche se più lento rispetto a UDP.

Porte riservate:

- **80**: HTTP di default
- **443**: HTTPS, versione sicura con SSL/TLS

HTTPS utilizza SSL/TLS per aggiungere sicurezza alle comunicazioni.

Messaggi HTTP: Richieste

Un messaggio HTTP è una richiesta o una risposta.

Richieste HTTP:

- **Request line**: metodo, risorsa, versione HTTP
- **Header**: impostazioni e metadati come host, user-agent, ecc.
- **Body**: dati specifici al metodo (es. dati di form per POST)

Esempio:

```
GET / HTTP/1.1
Host: localhost:8090
User-Agent: Mozilla/5.0
Accept: text/html,application/xhtml+xml
```

Messaggi HTTP: Risposte

Analogamente, le **risposte** HTTP sono composte da:

- **Response line**: versione HTTP, codice di stato
- **Header**: metadati come content-type e content-length
- **Body**: risorsa richiesta (HTML, immagine, JSON)

Esempio:

```
HTTP/1.1 200 OK
Content-Length: 662
Content-Type: text/html
Connection: close

<!DOCTYPE html>
...
```

Metodi HTTP

Metodi usati nel progetto:

- **GET**: Richiede una risorsa (pagina HTML, dati API)
- **POST**: Manda dati per creare una risorsa (es. form)
- **PUT**: Manda dati per modificare/creare una risorsa
- **DELETE**: Elimina dati dal server

Risposte di interesse:

- **200**: OK, richiesta processata correttamente
- **201**: CREATED, risorsa aggiunta
- **204**: NO CONTENT, processata senza body
- **400**: BAD REQUEST, richiesta malformata o incorretta

Server HTTP

Un server HTTP è un software che gestisce le richieste HTTP ricevute dai client (solitamente browser).

Viene utilizzato per:

- Hosting di siti web (pagine HTML)
- Gestione di API (servizi RESTful)
- Hosting di applicazioni web (codice eseguibile, es. JavaScript)

Lo scopo principale di un server HTTP è consegnare risorse attraverso il Web.

Metodologia ed Implementazione

Il server è stato sviluppato su ambiente Linux Ubuntu, utilizzando il linguaggio C e Git per il version control.

Il programma è suddiviso in moduli e compilato tramite Make.

Architettura:

- Quattro moduli principali: `main`, `response`, `request`, `keyvalue`.
- Cartella `www` contenente i documenti HTML da hostare.
- Makefile basilare per la compilazione.

Moduli:

- `main.c`: gestione TCP e creazione thread per connessioni.
- `keyvalue.h/c`: gestione coppie chiave-valore con mutex per evitare race condition.
- `request.h/c`: parsing delle richieste HTTP.
- `response.h/c`: creazione delle risposte HTTP e gestione delle operazioni.

Makefile

Makefile semplice per la compilazione:

```
CC = gcc
CFLAGS = -g -Wall

TARGET = httpserver
SRCS = main.c request.c keyvalue.c response.c

$(TARGET):
	$(CC) $(CFLAGS) -o $(TARGET) $(SRCS)


clean:
	rm httpserver
```

Il comando `make` compila il server, mentre `make clean` elimina l'eseguibile.

Implementazione del codice: Main

Il programma inizia configurando la connessione sulla porta 8090, mettendo il server in ascolto per connessioni in arrivo.

Per ogni connessione accettata, viene creato un thread che gestisce il client.

La funzione `handle_client` si occupa di:

- Ricevere la richiesta e inserirla in un buffer
- Effettuare il parsing della richiesta HTTP (`parse_http_request`)
- Gestire la richiesta e generare la risposta (`handle_request`)
- Inviare la risposta e chiudere la connessione

handle_client - Leggere le richieste del client

```c
// Gestione della richiesta di un client
void* handle_client(void *arg) {
    int client_fd = *(int *)arg;
    free(arg);

    char buffer[BUFFER_SIZE];
    ssize_t total = 0, n;
    while ((n = read(client_fd, buffer + total,
        BUFFER_SIZE - 1 - total)) > 0) {
         total += n;
         if (strstr(buffer, "\r\n\r\n")) break;
    }
    buffer[total] = '\0';

    printf("--- RAW ----------\n%s\n",buffer);
}
```

handle_client - Parsing richiesta HTTP

```
    HttpRequest nreq;
    parse_http_request(buffer, &nreq);

    printf("--- REQUEST --------------- \n%s\n%s\n%s\n%s
        \n%s\n",
            nreq.method, nreq.path, nreq.version, nreq.
                headers, nreq.body);
}
```

handle_client - Rispondi e chiudi

```
    char* resp = handle_request(&nreq);

    printf("--- RESPONSE ---------------\n%s\n", resp)
        ;

    if (send(client_fd, resp, strlen(resp), 0) == -1)
        {
         perror("Response send error");
         close(client_fd);
         exit(EXIT_FAILURE);
    }

    close(client_fd);
    pthread_exit(NULL);

    return NULL;
}
```

Modulo request: Struttura dati

Per gestire il parsing delle richieste HTTP, utilizziamo una struttura dati `HttpRequest` che contiene i campi principali:

```
typedef struct {
    char method[8];
    char path[256];
    char version[16];
    char headers[1024];
    char body[2048];
} HttpRequest;
```

Questa struttura permette di memorizzare le informazioni principali della richiesta per una facile gestione.

Parsing della request - Introduzione

- Parsing della request line (metodo, path, versione)
- Estrazione dell'header e del body tramite delimitatori `"\r\n\r\n"`
- Calcolo della lunghezza del body tramite il campo `Content-Length` negli header

parse_http_request - Parsing iniziale

```
//  inizializza la struttura dati
void parse_http_request(const char *raw_request,
    HttpRequest *req) {
     memset(req, 0, sizeof(HttpRequest));

     int parsed = sscanf(raw_request, "%7s %255s %15s",
          req->method, req->path, req->version);
     if (parsed != 3) {
          strcpy(req->method, "BAD");
          strcpy(req->path, "/");
          strcpy(req->version, "HTTP/1.0");
     }
}
```

parse_http_request - Estrazione header

```
const char *header_start = strstr(raw_request, "\r
    \n");
if (!header_start) return;
header_start += 2;

const char *body_start = strstr(raw_request, "\r\n
    \r\n");

if (body_start && (body_start > header_start)) {
    int header_len = (int)(body_start -
        header_start);
    if (header_len > 1023) header_len = 1023;
    strncpy(req->headers, header_start, header_len
        );
    req->headers[header_len] = '\0';
```

parse_http_request - Content-Length e body

```c
int content_length=0;
const char *cl=strcasestr(req->headers,"Content-Length:");
if(cl){
  cl+=strlen("Content-Length:");
  while(*cl==' ') cl++;
  content_length=atoi(cl);
}
if(content_length>0 && content_length<sizeof(req->body)){
  strncpy(req->body,body_start+4,content_length);
  req->body[content_length]='\0';
}else{
  req->body[0]='\0';
}
```

Modulo keyvalue: Gestione delle coppie chiave-valore

Lo struct KeyValue rappresenta una coppia chiave-valore. Le coppie sono memorizzate in un array globale protetto da un mutex per evitare race condition:

```
struct KeyValue {
    char key[64];
    char value[256];
};

extern struct KeyValue store[MAX_KV_PAIRS];
extern int store_count;
extern pthread_mutex_t store_mutex;
```

Le principali funzioni fornite sono:

- add_key_value – aggiunge una coppia
- is_kv_stored – verifica presenza della coppia
- delete_kv_pair – rimuove una coppia e riallinea l'array

Visualizzazione coppie chiave-valore in HTML (parte 1)

La funzione append_kv_to_html concatena a un documento HTML esistente la lista delle coppie chiave-valore, in modo da poterle visualizzare in una pagina web:

```
char *append_kv_to_html(const char *html) {
    char kv_html[2048];
    strcpy(kv_html, "<h2>Saved␣Pairs:</h2><ul>");

    pthread_mutex_lock(&store_mutex);
    for (int i = 0; i < store_count; i++) {
        strcat(kv_html, "<li>");
        strcat(kv_html, store[i].key);
        strcat(kv_html, ":␣");
        strcat(kv_html, store[i].value);
        strcat(kv_html, "</li>");
    }
```

Visualizzazione coppie chiave-valore in HTML (parte 2)

```
    }
    pthread_mutex_unlock(&store_mutex);

    strcat(kv_html, "</ul>");

    char *final = malloc(strlen(html) + strlen(kv_html
        ) + 1);
    strcpy(final, html);
    strcat(final, kv_html);

    return final;
}
```

Questa funzione garantisce l'accesso concorrente sicuro grazie al mutex.

Modulo response: funzione handle_request (parte 1)

La funzione principale del modulo response è handle_request, che soddisfa una richiesta e scrive la risposta.

```
char *handle_request(HttpRequest *req) {
    char *body = NULL;
    static char response[MAX_RESPONSE_SIZE];
    if (strcmp(req->method, "GET") == 0) {
        body = handle_get(req);
        write_http_response(response, 200, "text/html"
            , body);
    } else if (strcmp(req->method, "POST") == 0) {
        body = handle_post(req);
        write_http_response(response, 201, "text/html"
            , body);
    } else if (strcmp(req->method, "PUT") == 0) {
        int flg;
        body = handle_put(req, &flg);
        write_http_response(response, flg ? 201 : 204,
            "text/html", body);
```

Modulo response: funzione handle_request (parte 2)

```
    } else if (strcmp(req->method, "DELETE") == 0) {
        int flg;
        body = handle_delete(req, &flg);
        write_http_response(response, flg ? 201 : 204,
            "text/html", body);
    } else {
        body = strdup("<html><body><h1>400 Bad Request
            </h1></body></html>");
        write_http_response(response, 400, "text/html"
            , body);
    }

    if (body != NULL) {
        free(body);
    }

    return response;
}
```

Modulo `response`: descrizione e funzioni di supporto

- **GET**: ritorna il documento HTML corrispondente al path (200 OK).
- **POST**: aggiunge una coppia chiave-valore e ritorna il documento aggiornato (201 Created).
- **PUT**: aggiunge la coppia se non esiste (201 Created) o 204 No Content Se esiste.
- **DELETE**: elimina una coppia (201 Created) o 204 No Content Se non esiste.
- Altri metodi: risposta 400 Bad Request.

Funzioni di supporto:

- `get_html_file`: legge un file HTML e ne ritorna il contenuto.
- `parse_form_data`: interpreta il body con coppie chiave-valore, eseguendo operazioni (aggiungi, modifica, cancella).
- `write_http_response`: scrive la risposta HTTP completa.

Funzione write_http_response (parte 1)

Questa funzione scrive l'intestazione e il body di una risposta HTTP:

```
void write_http_response(char *response_buffer, int
    status_code,
    const char *content_type, const char *body) {

    const char *status_text;

    switch (status_code) {
        case 200: status_text = "OK"; break;
        case 201: status_text = "Created"; break;
        case 204: status_text = "No Content"; break;
        case 400: status_text = "Bad Request"; break;
        default: status_text = "Internal Server Error"
            ; break;
    }

    int content_length = body ? strlen(body) : 0;
```

Funzione write_http_response (parte 2)

```
    sprintf(response_buffer,
            "HTTP/1.1 %d %s\r\n"
            "Content-Length: %d\r\n"
            "Content-Type: %s\r\n"
            "Connection: close\r\n"
            "\r\n"
            "%s",
            status_code, status_text, content_length,
               content_type, body ? body : "");
}
```

Introduzione ai Test

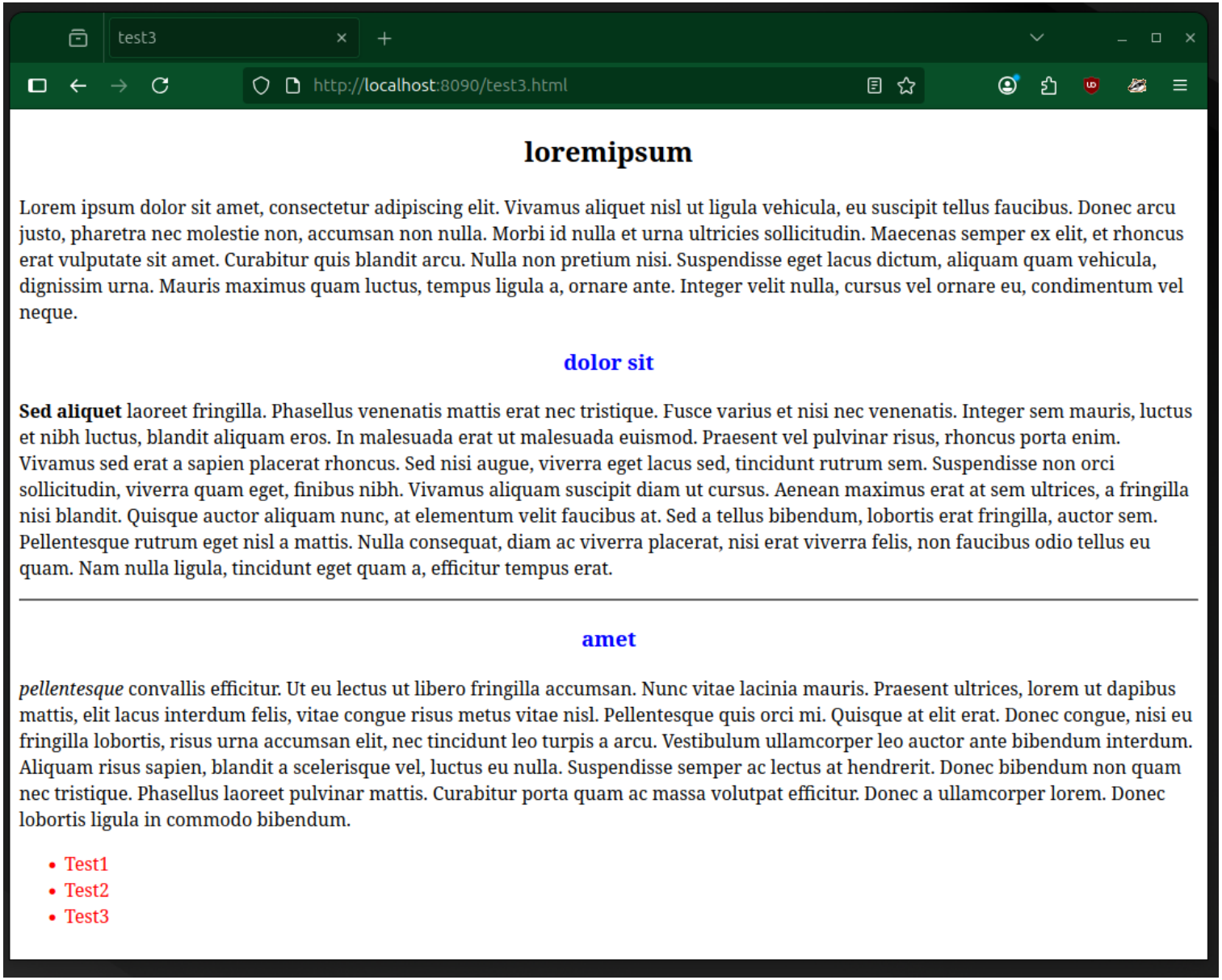
Per evidenziare i risultati ottenuti, verranno usati il browser *Firefox* e il comando *curl* per inviare richieste al server.

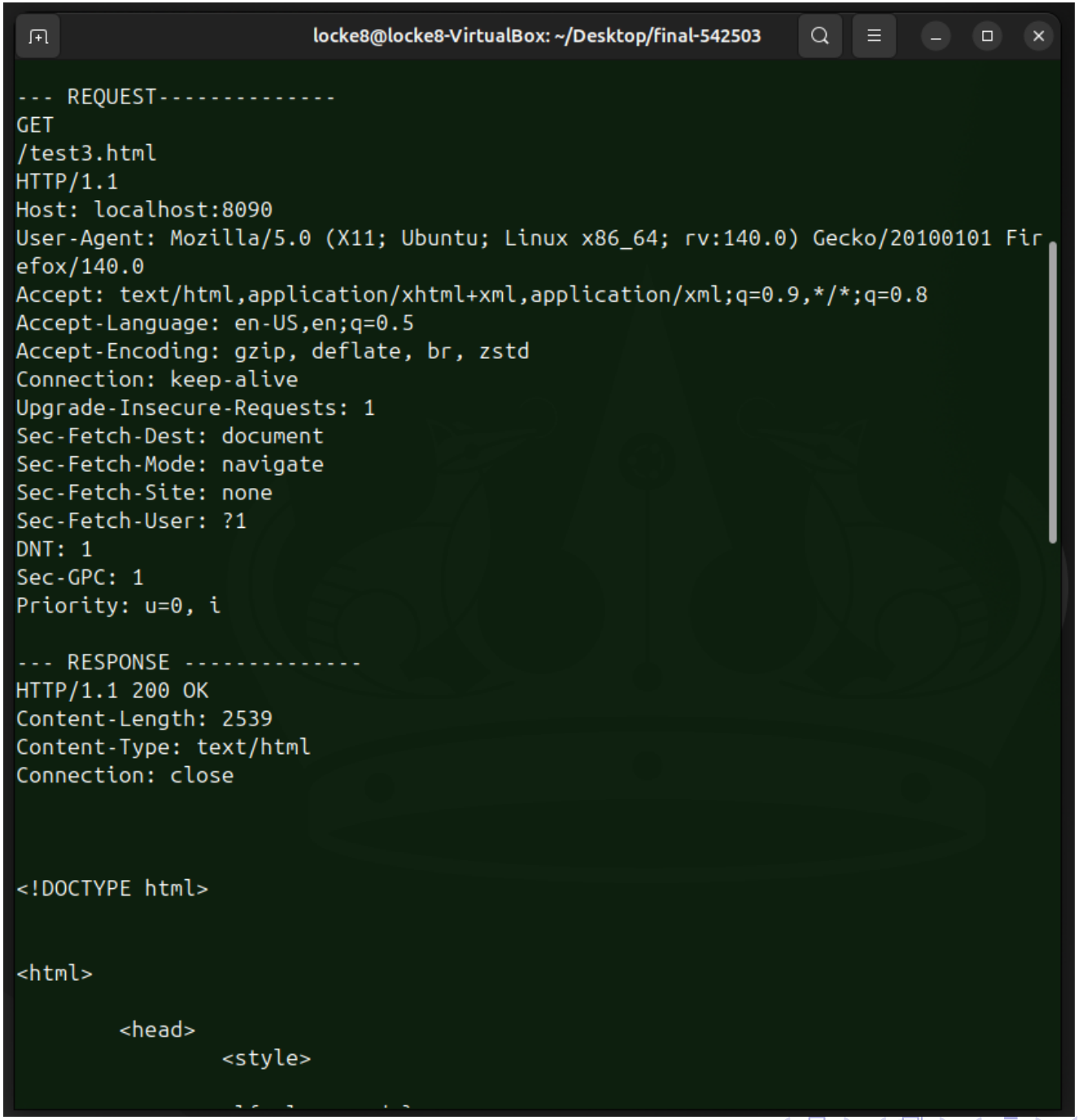
Tipologie di richieste testate:

- GET
- POST
- PUT
- DELETE
- BAD REQUEST
- Test di concorrenza

Test: GET

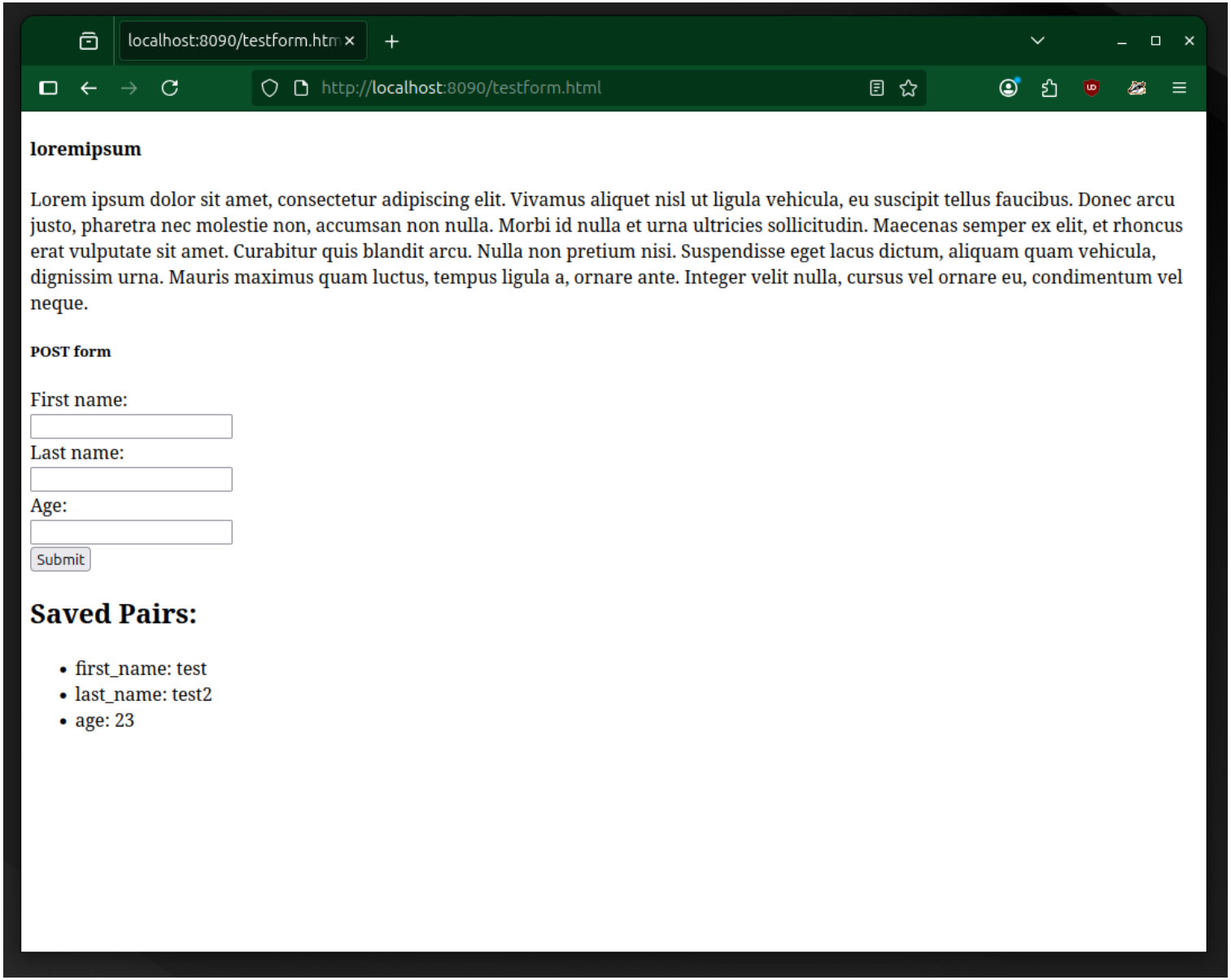
- Per testare la richiesta GET, basta collegarsi via browser al server.
- Il server invia correttamente il documento HTML con codice 200 (OK).

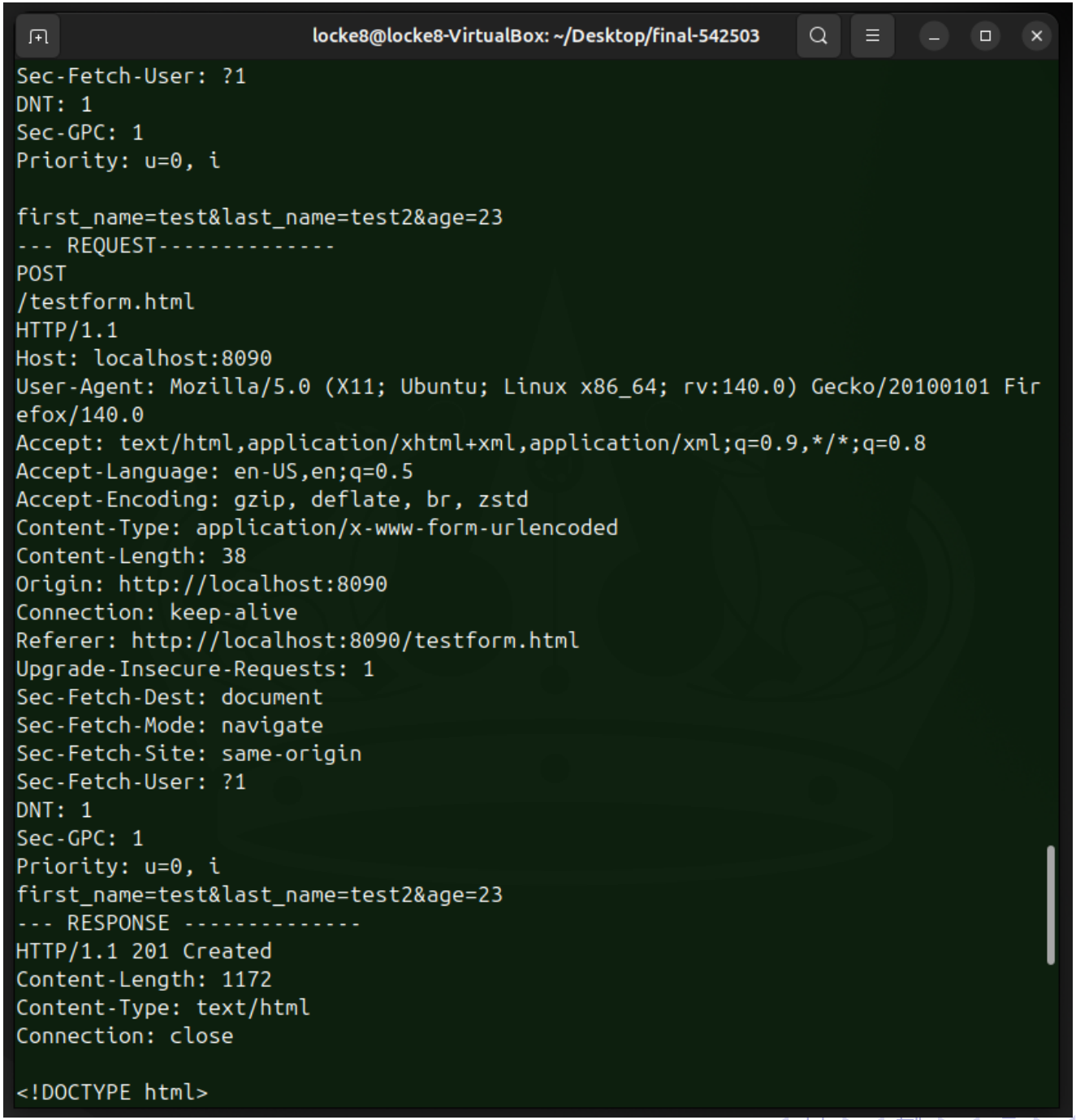

test3
http://localhost:8090/test3.html
loremipsum
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vivamus aliquet nisl ut ligula vehicula, eu suscipit tellus faucibus. Donec arcu justo, pharetra nec molestie non, accumsan non nulla. Morbi id nulla et urna ultricies sollicitudin. Maecenas semper ex elit, et rhoncus erat vulputate sit amet. Curabitur quis blandit arcu. Nulla non pretium nisi. Suspendisse eget lacus dictum, aliquam quam vehicula, dignissim urna. Mauris maximus quam luctus, tempus ligula a, ornare ante. Integer velit nulla, cursus vel ornare eu, condimentum vel neque.
dolor sit
Sed aliquet laoreet fringilla. Phasellus venenatis mattis erat nec tristique. Fusce varius et nisi nec venenatis. Integer sem mauris, luctus et nibh luctus, blandit aliquam eros. In malesuada erat ut malesuada euismod. Praesent vel pulvinar risus, rhoncus porta enim. Vivamus sed erat a sapien placerat rhoncus. Sed nisi augue, viverra eget lacus sed, tincidunt rutrum sem. Suspendisse non orci sollicitudin, viverra quam eget, finibus nibh. Vivamus aliquam suscipit diam ut cursus. Aenean maximus erat at sem ultrices, a fringilla nisi blandit. Quisque auctor aliquam nunc, at elementum velit faucibus at. Sed a tellus bibendum, lobortis erat fringilla, auctor sem. Pellentesque rutrum eget nisl a mattis. Nulla consequat, diam ac viverra placerat, nisi erat viverra felis, non faucibus odio tellus eu quam. Nam nulla ligula, tincidunt eget quam a, efficitur tempus erat.
amet
pellentesque convallis efficitur. Ut eu lectus ut libero fringilla accumsan. Nunc vitae lacinia mauris. Praesent ultrices, lorem ut dapibus mattis, elit lacus interdum felis, vitae congue risus metus vitae nisl. Pellentesque quis orci mi. Quisque at elit erat. Donec congue, nisi eu fringilla lobortis, risus urna accumsan elit, nec tincidunt leo turpis a arcu. Vestibulum ullamcorper leo auctor ante bibendum interdum. Aliquam risus sapien, blandit a scelerisque vel, luctus eu nulla. Suspendisse semper ac lectus at hendrerit. Donec bibendum non quam nec tristique. Phasellus laoreet pulvinar mattis. Curabitur porta quam ac massa volutpat efficitur. Donec a ullamcorper lorem. Donec lobortis ligula in commodo bibendum.
Test1
Test2
Test3


locke8@locke8-VirtualBox: ~/Desktop/final-542503
--- REQUEST-------------
GET
/test3.html
HTTP/1.1
Host: localhost:8090
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:140.0) Gecko/20100101 Fir
efox/140.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br, zstd
Connection: keep-alive
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: none
Sec-Fetch-User: ?1
DNT: 1
Sec-GPC: 1
Priority: u=0, i
--- RESPONSE -------------
HTTP/1.1 200 OK
Content-Length: 2539
Content-Type: text/html
Connection: close
<!DOCTYPE html>
<html>
<head>
<style>

Test: POST

- Usata pagina HTML con form `method=POST`.
- Il server aggiunge la coppia chiave-valore e risponde con codice 201 (CREATED).


localhost:8090/testform.htm
http://localhost:8090/testform.html
loremipsum
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vivamus aliquet nisl ut ligula vehicula, eu suscipit tellus faucibus. Donec arcu justo, pharetra nec molestie non, accumsan non nulla. Morbi id nulla et urna ultricies sollicitudin. Maecenas semper ex elit, et rhoncus erat vulputate sit amet. Curabitur quis blandit arcu. Nulla non pretium nisi. Suspendisse eget lacus dictum, aliquam quam vehicula, dignissim urna. Mauris maximus quam luctus, tempus ligula a, ornare ante. Integer velit nulla, cursus vel ornare eu, condimentum vel neque.
POST form
First name:
Last name:
Age:
Submit
Saved Pairs:
first_name: test
last_name: test2
age: 23


locke8@locke8-VirtualBox: ~/Desktop/final-542503
Sec-Fetch-User: ?1
DNT: 1
Sec-GPC: 1
Priority: u=0, i
first_name=test&last_name=test2&age=23
--- REQUEST--------------
POST
/testform.html
HTTP/1.1
Host: localhost:8090
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:140.0) Gecko/20100101 Fir
efox/140.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br, zstd
Content-Type: application/x-www-form-urlencoded
Content-Length: 38
Origin: http://localhost:8090
Connection: keep-alive
Referer: http://localhost:8090/testform.html
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: same-origin
Sec-Fetch-User: ?1
DNT: 1
Sec-GPC: 1
Priority: u=0, i
first_name=test&last_name=test2&age=23
--- RESPONSE --------------
HTTP/1.1 201 Created
Content-Length: 1172
Content-Type: text/html
Connection: close
<!DOCTYPE html>

Test: PUT

- Comando curl per inviare una richiesta PUT:

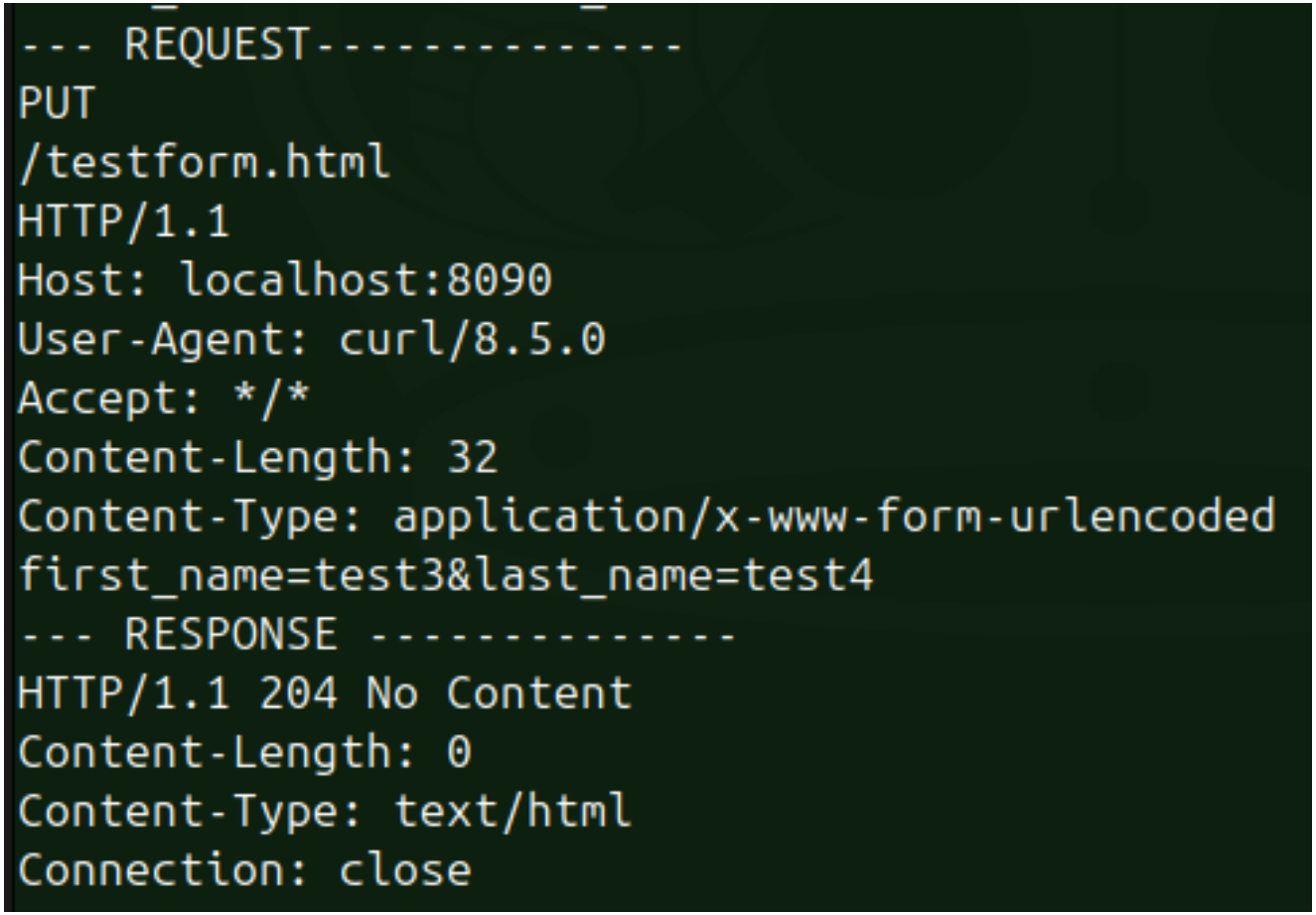
```
curl -X PUT http://localhost:8090/testform.html \
     -d "first_name=test3&last_name=test4"
```

- Il server aggiunge la coppia e risponde 201 (CREATED).
- Se la stessa richiesta viene inviata di nuovo, il server risponde 204 (NO CONTENT).

locke8@locke8-VirtualBox: ~/Desktop/final-542503

```
Priority: u=0, i

first_name=test&last_name=test2&age=23
--- REQUEST--------------
PUT
/testform.html
HTTP/1.1
Host: localhost:8090
User-Agent: curl/8.5.0
Accept: */*
Content-Length: 39
Content-Type: application/x-www-form-urlencoded
first_name=test3&last_name=test4&age=36
--- RESPONSE --------------
HTTP/1.1 201 Created
Content-Length: 1239
Content-Type: text/html
Connection: close

<!DOCTYPE html>

<html>

        <head>
        </head>

        <body>

                <h4>loremipsum</h4>

                <p>
                Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vivamus
 aliquet nisl ut ligula vehicula, eu suscipit tellus faucibus. Donec arcu justo,
 pharetra nec molestie non, accumsan non nulla. Morbi id nulla et urna ultricies
 sollicitudin. Maecenas semper ex elit, et rhoncus erat vulputate sit amet. Cura
bitur quis blandit arcu. Nulla non pretium nisi. Suspendisse eget lacus dictum,
aliquam quam vehicula, dignissim urna. Mauris maximus quam luctus, tempus ligula
```

```
<h2>Saved Pairs:</h2><ul><li>first_name: test</li><li>last_name: test2</li><li>a
ge: 23</li><li>first_name: test3</li><li>last_name: test4</li><li>age: 36</li></
ul>
```


--- REQUEST-------------
PUT
/testform.html
HTTP/1.1
Host: localhost:8090
User-Agent: curl/8.5.0
Accept: */*
Content-Length: 32
Content-Type: application/x-www-form-urlencoded
first_name=test3&last_name=test4
--- RESPONSE -------------
HTTP/1.1 204 No Content
Content-Length: 0
Content-Type: text/html
Connection: close

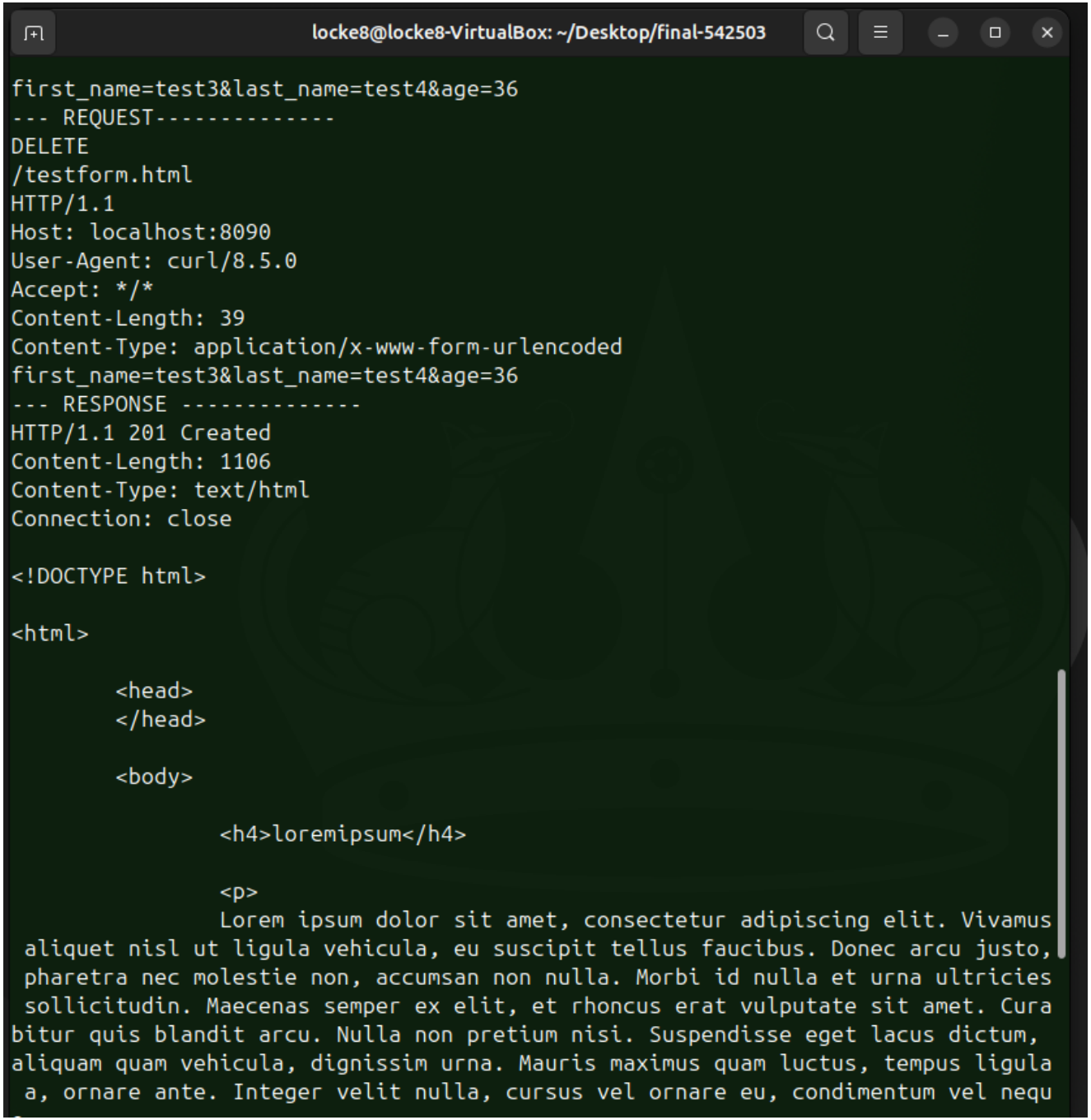
Test: DELETE

- Prima si aggiungono le coppie con PUT:

```
curl -X PUT http://localhost:8090/testform.html \
     -d "first_name=test3&last_name=test4"
```

- Poi si eliminano con DELETE:

```
curl -X DELETE http://localhost:8090/testform.html \
     -d "first_name=test3&last_name=test4"
```


locke8@locke8-VirtualBox: ~/Desktop/final-542503
first_name=test3&last_name=test4&age=36
--- REQUEST--------------
DELETE
/testform.html
HTTP/1.1
Host: localhost:8090
User-Agent: curl/8.5.0
Accept: */*
Content-Length: 39
Content-Type: application/x-www-form-urlencoded
first_name=test3&last_name=test4&age=36
--- RESPONSE --------------
HTTP/1.1 201 Created
Content-Length: 1106
Content-Type: text/html
Connection: close
<!DOCTYPE html>
<html>
<head>
</head>
<body>
<h4>loremipsum</h4>
<p>
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vivamus
aliquet nisl ut ligula vehicula, eu suscipit tellus faucibus. Donec arcu justo,
pharetra nec molestie non, accumsan non nulla. Morbi id nulla et urna ultricies
sollicitudin. Maecenas semper ex elit, et rhoncus erat vulputate sit amet. Cura
bitur quis blandit arcu. Nulla non pretium nisi. Suspendisse eget lacus dictum,
aliquam quam vehicula, dignissim urna. Mauris maximus quam luctus, tempus ligula
a, ornare ante. Integer velit nulla, cursus vel ornare eu, condimentum vel nequ

```
<h2>Saved Pairs:</h2><ul></ul>
```

```
--- REQUEST-------------
DELETE
/testform.html
HTTP/1.1
Host: localhost:8090
User-Agent: curl/8.5.0
Accept: */*
Content-Length: 39
Content-Type: application/x-www-form-urlencoded
first_name=test3&last_name=test4&age=36
--- RESPONSE --------------
HTTP/1.1 204 No Content
Content-Length: 0
Content-Type: text/html
Connection: close
```

Test: BAD REQUEST

- Usando curl con metodo non gestito (OPTIONS):

```
curl -X OPTIONS http://localhost:8090/testform.html
```

```
--- REQUEST--------------
OPTIONS
/testform.html
HTTP/1.1
Host: localhost:8090
User-Agent: curl/8.5.0
Accept: */*

--- RESPONSE --------------
HTTP/1.1 400 Bad Request
Content-Length: 50
Content-Type: text/html
Connection: close

<html><body><h1>400 Bad Request</h1></body></html>
```

Test di Concorrenza

Tre richieste inviate simultaneamente con curl:

```
curl -X POST http://localhost:8090/testform.html -d "
    first_name=test1&last_name=test2&age=22" &
curl -X PUT http://localhost:8090/testform.html -d "
    first_name=test1&last_name=test2&age=22" &
curl -X DELETE http://localhost:8090/testform.html -d
    "first_name=test1&last_name=test2&age=22" &
wait
```

Risposte ricevute:

- DELETE: 204 NO CONTENT (coppia non presente)
- POST: 201 CREATED (aggiunta coppie)
- PUT: 204 NO CONTENT (coppie già presenti)

```
--- REQUEST--------------
DELETE
/testform.html
HTTP/1.1
Host: localhost:8090
User-Agent: curl/8.5.0
Accept: */*
Content-Length: 39
Content-Type: application/x-www-form-urlencoded
first_name=test1&last_name=test2&age=22
--- RESPONSE --------------
HTTP/1.1 204 No Content
Content-Length: 0
Content-Type: text/html
Connection: close
```

```
--- REQUEST--------------
POST
/testform.html
HTTP/1.1
Host: localhost:8090
User-Agent: curl/8.5.0
Accept: */*
Content-Length: 39
Content-Type: application/x-www-form-urlencoded
first_name=test1&last_name=test2&age=22
--- RESPONSE --------------
HTTP/1.1 201 Created
Content-Length: 1173
Content-Type: text/html
Connection: close

<!DOCTYPE html>
```

```
<h2>Saved Pairs:</h2><ul><li>first_name: test1</li><li>last_name: te
st2</li><li>age: 22</li></ul>
```

```
--- REQUEST--------------
PUT
/testform.html
HTTP/1.1
Host: localhost:8090
User-Agent: curl/8.5.0
Accept: */*
Content-Length: 39
Content-Type: application/x-www-form-urlencoded
first_name=test1&last_name=test2&age=22
--- RESPONSE --------------
HTTP/1.1 204 No Content
Content-Length: 0
Content-Type: text/html
Connection: close
```

Conclusione

Il server sviluppato è capace di gestire le richieste principali del protocollo HTTP e di fornire risposte soddisfacenti. Grazie alle capacità di concorrenza, il server riesce a gestire molteplici clienti in maniera efficiente.

Il codice è inoltre una solida base di sviluppo, estendibile facilmente grazie alla struttura modulare.

Sviluppi Futuri

Alcune possibili estensioni del server includono:

- Supportare ulteriori metodi HTTP (`HEAD`, `OPTIONS`)
- Gestire connessioni con **keep-alive** per mantenere aperta la connessione dopo la risposta
- Adottare la versione sicura HTTPS
- Inviare coppie chiave-valore a CGI per eseguire programmi esterni e creare contenuti dinamici