Università degli Studi di Messina

CdL in Scienze Informatiche
Reti di calcolatori e sistemi distribuiti - Modulo B
A.A. 2024/2025

Realizzazione di un Server REST concorrente in C con backend Sqlite

Relazione a cura di:

Sandi Russo 553675

Sommario

La presente relazione descrive la progettazione e l'implementazione di un server RESTful concorrente, scritto interamente in linguaggio C. L'obiettivo del progetto è stato quello di sviluppare un server web leggero e performante, capace di gestire richieste API REST per operazioni CRUD (Create, Read, Update, Delete) su una risorsa "utenti", con persistenza dei dati garantita da un backend **SQLite**.

Per affrontare la necessità di servire un elevato numero di client simultaneamente, è stato adottato un modello di concorrenza avanzato, che si discosta dal tradizionale "un thread per client". L'architettura si basa su un design event-driven che utilizza il multiplexing di I/O non bloccante tramite **epoll** (Linux API). Un thread principale gestisce tutti gli eventi di rete, accettando nuove connessioni e leggendo le richieste. L'elaborazione effettiva delle richieste (parsing HTTP, logica di business e query al database) è delegata a un **Thread Pool** di worker pre-allocati. Questo approccio, noto come modello Reactor/Proactor, minimizza l'overhead di creazione e distruzione dei thread, massimizzando la scalabilità e la reattività del server.

Indice

1 Introduzione

1.1 Contesto e Obiettivi

Nel panorama moderno dello sviluppo software, le API (Application Programming Interface) rappresentano il collante fondamentale che permette a sistemi distribuiti e microservizi di comunicare. Tra i vari stili architetturali per la creazione di API, **REST** (Representational State Transfer) si è imposto come standard de-facto per la sua semplicità, scalabilità e l'utilizzo dei protocolli standard del web, primo tra tutti l'HTTP.

Un'API RESTful modella le operazioni su entità di business come "risorse" (ad esempio, un utente) che possono essere manipolate tramite i metodi standard del protocollo HTTP:

- **GET**: per recuperare una risorsa.
- **POST**: per creare una nuova risorsa.
- **PUT/PATCH**: per aggiornare una risorsa esistente.
- **DELETE**: per eliminare una risorsa.

L'obiettivo di questo progetto è stata la realizzazione *from-scratch* di un server REST concorrente in C. La scelta del C, sebbene più complessa rispetto a linguaggi di alto livello, è motivata dalla volontà di ottenere il massimo controllo sulle risorse di sistema, sulla gestione della memoria e sulle performance di rete, realizzando un servizio estremamente leggero ed efficiente. Tale approccio permette inoltre di comprendere a fondo i meccanismi interni che regolano la comunicazione web e la gestione della concorrenza a basso livello, spesso astratti dai framework più moderni.

Gli obiettivi specifici del progetto sono:

1. Implementare un server HTTP basilare in grado di ricevere e rispondere a richieste.
2. Gestire le richieste in modo concorrente e scalabile, utilizzando I/O non bloccante (`epoll`) e un thread pool.
3. Realizzare un'API REST per le operazioni CRUD su una risorsa "utenti".
4. Integrare un backend di database **SQLite** per la persistenza dei dati.

2 Definizione del Problema

2.1 Analisi del Protocollo HTTP e REST

A differenza di protocolli più complessi come FTP (che richiede due canali distinti), l'architettura REST si appoggia interamente sul protocollo HTTP, un protocollo di richiesta-risposta testuale e senza stato (*stateless*).

Ogni interazione è composta da:

- **Richiesta (Request)**: Inviata dal client al server. Contiene un *metodo* (es. GET), un *percorso* (es. /users/1), *header* (metadati) e un eventuale *body* (corpo della richiesta, es. per POST).
- **Risposta (Response)**: Inviata dal server al client. Contiene un *codice di stato* (es. 200 OK, 404 Not Found), *header* e un eventuale *body* (es. i dati richiesti in formato JSON).

Un server REST non fa altro che interpretare queste richieste HTTP, mappare il metodo e il percorso a un'azione specifica (es. una funzione C) ed eseguire tale azione (spesso interagendo con un database), per poi formattare una risposta HTTP adeguata.

2.2 Requisiti di Concorrenza: Il Modello Epoll + Thread Pool

Un server di rete robusto deve gestire migliaia di connessioni simultanee (il cosiddetto problema C10k). Il modello "un thread per client" (utilizzato nell'esempio FTP) è semplice da implementare ma non scala: ogni thread consuma risorse (memoria per lo stack) e il context switching tra migliaia di thread diventa un collo di bottiglia.

Per questo progetto è stata scelta un'architettura molto più performante, basata su due pilastri:

1. **I/O Multiplexing con `epoll` (Reactor)**: Il thread principale (il *Reactor*) utilizza `epoll` per monitorare in modo efficiente e non bloccante migliaia di socket contemporaneamente. Invece di attendere (bloccarsi) su un singolo `read()` o `accept()`, il thread attende su `epoll_wait()`, che lo risveglia solo quando c'è effettivamente un evento (una nuova connessione, o dati da leggere) su *uno qualsiasi* dei socket monitorati.
2. **Thread Pool (Proactor)**: L'elaborazione delle richieste può essere costosa (parsing, query al database). Se il thread principale di `epoll` si bloccasse per eseguire una query, tutte le altre connessioni rimarrebbero in attesa. Per evitare ciò, il thread principale delega il lavoro pesante. Quando `epoll` notifica una nuova richiesta, il thread principale si limita a leggerla e a impacchettarla come un "lavoro" (*job*) che viene inserito in una coda. Un pool di **worker thread**, pre-allocati all'avvio (*Proactors*), attende su questa coda. Non appena un lavoro è disponibile, un thread libero lo preleva e lo esegue (parsing HTTP, query SQLite, invio risposta), tornando poi in attesa di nuovi lavori.

Questo design separa nettamente la gestione (veloce) degli eventi I/O dal processamento (potenzialmente lento) della logica applicativa, garantendo massima reattività e scalabilità.

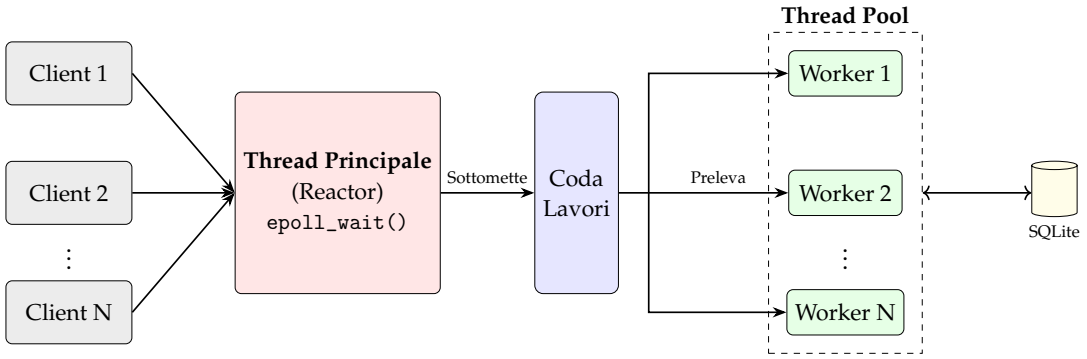

Figura 1: Architettura del server: Epoll + Thread Pool

2.3 Requisiti Funzionali del Server

Per dimostrare il funzionamento dell'architettura, il server implementa un'API REST basilare per la gestione di una risorsa `users`. Le operazioni (endpoint) supportate sono:

Tabella 1: Endpoint dell'API REST implementati

Metodo	Percorso (Path)	Scopo
`GET`	`/users`	Recupera l'elenco di tutti gli utenti.
`GET`	`/users/{id}`	Recupera i dettagli di un singolo utente.
`POST`	`/users`	Crea un nuovo utente. Il nome è passato nel body.
`DELETE`	`/users/{id}`	Elimina un utente specifico.

Tutti i dati sono resi persistenti su un database **SQLite**, memorizzato nel file `rest_api.db`. Il server crea automaticamente la tabella `users` al primo avvio, se non esiste.

3 Metodologia

Il progetto è stato suddiviso in moduli C, ognuno con una responsabilità specifica, per garantire manutenibilità e chiara separazione dei compiti.

3.1 Struttura del Progetto e Flusso di Esecuzione

3.1.1 Il Punto di Ingresso: `server.c`

Il file `main.c` è responsabile dell'inizializzazione e del ciclo di vita del server (il Reactor). Le fasi principali sono:

1. Inizializzazione del database (`db_init()`).
2. Creazione del thread pool (`threadpool_create()`).
3. Creazione e setup del socket di ascolto (`socket`, `bind`, `listen`).
4. Creazione dell'istanza `epoll` (`epoll_create1()`).
5. Aggiunta del socket di ascolto a `epoll` (`epoll_ctl_add`).

Il cuore del server è il loop `while(1)` che attende eventi su `epoll_wait()`.

Questo blocco di codice mostra la prima metà del loop eventi. Il server attende su `epoll_wait()` un evento. Se l'evento notificato appartiene al socket `server_fd` principale, significa che è arrivata una nuova connessione. Il server quindi la accetta (`accept`), la imposta come non bloccante e la aggiunge all'istanza di `epoll` per monitorare futuri eventi di lettura.

```
while (1) {
    int n = epoll_wait(epoll_fd, events, MAX_EVENTS, -1);
    // ... gestione errori ...

    for (int i = 0; i < n; i++) {

        // evento sul socket di ascolto -> Nuova connessione
        if (events[i].data.fd == server_fd) {
            int client_fd = accept(server_fd, ...);
            // ... gestione errori ...
            set_nonblocking(client_fd);

            ev.events = EPOLLIN | EPOLLET | EPOLLONESHOT;
            ev.data.fd = client_fd;
            epoll_ctl(epoll_fd, EPOLL_CTL_ADD, client_fd, &ev)
;
        }
```

Listing 1: **server.c** - Loop eventi: gestione nuove connessioni (1/2)

Questa è la seconda metà del loop `epoll`. Se l'evento non è una nuova connessione (blocco `else`), allora deve essere un evento di lettura su un client già connesso. Il thread principale (il Reactor) non esegue il lavoro direttamente; se lo facesse, operazioni potenzialmente bloccanti come il `read()` o le query al database congelerebbero l'intero server, impedendogli di gestire altri client. Prepara invece una struttura `RequestContext` che "impacchetta" tutte le informazioni necessarie (il file descriptor del client, l'istanza di epoll e la connessione al database) e la "sottomette" alla coda del thread pool tramite `threadpool_add_job()`. Questo passaggio segnala a un thread worker in attesa di svegliarsi, prendere in carico la richiesta ed eseguirla in parallelo, lasciando il thread principale immediatamente libero di tornare in `epoll_wait` e servire nuove richieste. L'uso di `EPOLLONESHOT` (impostato alla `accept`) è fondamentale in questo design, poiché impedisce ad `epoll` di notificare nuovi eventi sullo stesso socket mentre un worker lo sta già processando, evitando così race condition.

```
1           // evento su un client socket -> Dati da leggere
2           else {
3               // prepara il contesto per il worker
4               RequestContext* ctx = malloc(sizeof(RequestContext
    ));
5               ctx->client_fd = events[i].data.fd;
6               ctx->epoll_fd = epoll_fd;
7               ctx->db = db;
8
9               // sottomette il lavoro al pool
10              threadpool\_add\_job(pool, handle\_http\_request,
    ctx);
11          }
12      }
13  }
```

Listing 2: **server.c** - Loop eventi: sottomissione lavori (2/2)

3.1.2 Gestione della Concorrenza: `thread_pool.c`

Questo modulo implementa un classico thread pool. La funzione `worker_thread` è l'implementazione di ogni thread "operaio".

Ogni worker attende su una *condition variable* finché la coda dei lavori non è vuota. Quando un nuovo lavoro è aggiunto (`threadpool_add_job`), la *condition variable* viene segnalata, un worker si sveglia, estrae il lavoro dalla coda (protetto da `pthread_mutex`) ed esegue la funzione.

La prima parte della funzione `worker_thread` mostra come il thread gestisce l'attesa. Acquisisce un `mutex` per proteggere la coda dei lavori. Entra quindi in un ciclo `while` che, se la coda è vuota e non è in corso uno spegnimento, invoca `pthread_cond_wait()`. Questa funzione rilascia atomicamente il mutex e mette il thread in sonno, in attesa che un nuovo lavoro venga aggiunto.

```c
static void* worker_thread(void* arg) {
    ThreadPool* pool = (ThreadPool*)arg;

    while (1) {
        pthread_mutex_lock(&pool->mutex);

        // aspetta fino a quando ci sono job, o e' richiesto lo shutdown
        while (pool->queue_head == NULL && !pool->shutdown) {
            pthread_cond_wait(&pool->cond, &pool->mutex);
        }

        // se e' shutdown e la coda e' vuota, esci
        if (pool->shutdown && pool->queue_head == NULL) {
            pthread_mutex_unlock(&pool->mutex);
            break;
        }
```

Listing 3: **thread_pool.c** - Worker: attesa di un lavoro (1/2)

Una volta che il thread si riattiva (e ri-acquisisce il mutex), esce dal ciclo di attesa. Estrae il primo lavoro (`JobNode`) dalla testa della coda. Un dettaglio cruciale è che rilascia il `mutex` *prima* di eseguire il lavoro (`job->function(job->arg)`). Questo permette agli altri worker di accedere alla coda mentre questo thread è occupato, massimizzando la concorrenza del pool

```c
        // prendo il primo job dalla coda
        JobNode* job = pool->queue_head;
        if (job != NULL) {
            pool->queue_head = job->next;
            if (pool->queue_head == NULL) {
                pool->queue_tail = NULL;
            }
        }
        // rilascio il mutex prima di eseguire il job
        pthread_mutex_unlock(&pool->mutex);

        // eseguo il job
        if (job != NULL) {
            job->function(job->arg);
            free(job);
        }
    }
    return NULL;
}
```

Listing 4: **thread_pool.c** - Worker: esecuzione del lavoro (2/2)

3.2 Gestione delle Richieste HTTP: `http_handler.c`

Questa è la funzione che viene eseguita da un worker thread. È il cuore della logica applicativa ed è stata suddivisa in tre fasi principali: preparazione, routing ed invio.

Questa è la prima fase eseguita da un worker thread. Il codice legge i dati grezzi della richiesta HTTP dal `client_fd` e li memorizza in un buffer. Utilizza `sscanf` per un parsing basilare, estraendo il metodo (es. `GET`) e il percorso (es. `/users`). Infine, inizializza le variabili di default per la risposta (es. `200 OK`).

```
void handle_http_request(void* arg) {
    RequestContext* ctx = (RequestContext*)arg;
    char buffer[BUFFER_SIZE] = {0};

    // legge la richiesta
    ssize_t bytes_read = read(ctx->client_fd, buffer,
                              BUFFER_SIZE - 1);

    if (bytes_read <= 0) {
        // ... gestione disconnessione ...
        goto cleanup;
    }

    // parsing della richiesta
    char method[16], path[256];
    if (sscanf(buffer, "%15s %255s", method, path) != 2) {
        send_response(ctx->client_fd, 400, "Bad Request", ...)
    ;
        goto cleanup;
    }

    // setup variabili per la risposta
    char* response_body = NULL;
    int status_code = 200;
    const char* status_text = "OK";
    const char* content_type = "application/json";
```

Listing 5: **http_handler.c** - Fase 1: Lettura e Parsing (1/3)

Questo blocco costituisce il "router" dell'applicazione. Attraverso una catena di `if-else if`, confronta il metodo e il percorso parsificati. Se trova una corrispondenza (es. `GET /users`), invoca la funzione appropriata del gestore database (es. `db_get_all_users`). Questa è una forma di dispatching manuale che separa nettamente la logica HTTP dalla logica di accesso ai dati; infatti, il router non sa come il database funziona, ma solo quale funzione chiamare. Allo stesso modo, una richiesta `POST /users` viene instradata a `db_create_user`, mentre un `DELETE` su `/users/id` richiama `db_delete_user`. Il router è anche responsabile di mappare il risultato di queste operazioni ai corretti codici di stato HTTP, impostando `404 Not Found` per risorse non trovate o `201 Created` per una creazione avvenuta con successo.

```c
    // esempio: GET /users
    if (strcmp(method, "GET") == 0 &&
        strcmp(path, "/users") == 0) {
        response_body = db_get_all_users(ctx->db);
    } // esempio: GET /users/<id>
    else if (strcmp(method, "GET") == 0 &&
             strncmp(path, "/users/", 7) == 0) {
        int user_id = atoi(path + 7);
        response_body = db_get_user(ctx->db, user_id);
        if (!response_body) {
            status_code = 404;
            status_text = "Not Found";
        }
    } // esempio: POST /users
    else if (strcmp(method, "POST") == 0 &&
             strcmp(path, "/users") == 0) {
        char* body = extract_body(buffer);
        if (body && db_create_user(ctx->db, body)) {
            status_code = 201;
            status_text = "Created";
        } else {
            status_code = 400;
            status_text = "Bad Request";
        }
    }
```

Listing 6: **http_handler.c** - Fase 2: Routing Logico (2/3)

Nella fase finale, la funzione `send_response` (non mostrata) viene chiamata per costruire e inviare la risposta HTTP completa al client. Dopo l'invio, la memoria allocata per il corpo della risposta viene liberata. Infine, l'etichetta `cleanup` gestisce la chiusura della connessione: il file descriptor viene rimosso dall'istanza `epoll` (`EPOLL_CTL_DEL`) e il socket viene chiuso.

```c
    // invia la risposta al client
    send_response(ctx->client_fd, status_code, status_text,
                  content_type, response_body);

    if (response_body) free(response_body);

cleanup:
    // rimuove il fd da epoll e chiude la connessione
    epoll_ctl(ctx->epoll_fd, EPOLL_CTL_DEL,
              ctx->client_fd, NULL);
    close(ctx->client_fd);
    free(ctx);
}
```

Listing 7: **http_handler.c** - Fase 3: Risposta e Cleanup (3/3)

3.3 Interfacciamento al Database: `db_handler.c`

Questo modulo agisce da Data Access Layer (DAL), incapsulando tutta la logica di interazione con SQLite e astraendola dal gestore HTTP. Questo frammento gestisce la richiesta per ottenere tutti gli utenti. La query SQL viene preparata in modo sicuro tramite `sqlite3_prepare_v2`, che previene attacchi di tipo SQL injection. Successivamente, viene allocato un buffer per contenere la stringa JSON di risposta e viene aggiunto il carattere di apertura dell'array (`[`).

```
char* db_get_all_users(sqlite3* db) {
    sqlite3_stmt* stmt;
    const char* sql = "SELECT id, name FROM users;";
    if (sqlite3_prepare_v2(db, sql, -1, &stmt, NULL) !=
   SQLITE_OK) {
        fprintf(stderr, "Failed to prepare statement: %s\n",
                sqlite3_errmsg(db));
        return NULL;
    }
    // alloca buffer e inizia a costruire il JSON
    char* json_result = (char*)calloc(BUFFER_SIZE * 2,
                                      sizeof(char));
    strcat(json_result, "[");
    int first_row = 1;
```

Listing 8: **db_handler.c** - Read (GET /users): preparazione query (1/2)

Qui, il codice itera su ogni riga restituita dalla query tramite un ciclo `while` su `sqlite3_step()`. Per ogni riga (utente), formatta i dati in una stringa JSON (es. `{"id":1, "name":"Sandi"}`) usando `sprintf` e la concatena al risultato. Infine, aggiunge la parentesi chiusa (`]`) e rilascia le risorse della query.

```
    // itera sui risultati
    while (sqlite3_step(stmt) == SQLITE_ROW) {
        if (!first_row) strcat(json_result, ",");

        char user_json[512];
        sprintf(user_json, "{\"id\":%d, \"name\":\"%s\"}",
                sqlite3_column_int(stmt, 0),
                sqlite3_column_text(stmt, 1));

        // concatena il JSON dell'utente al risultato
        strcat(json_result, user_json);
        first_row = 0;
    }
    strcat(json_result, "]");
    sqlite3_finalize(stmt);
    return json_result;}
```

Listing 9: **db_handler.c** - Read (GET /users): costruzione JSON (2/2)

Questa funzione gestisce la creazione di un nuovo utente. Come prima, prepara una query SQL parametrizzata (`INSERT ... VALUES (?)`). Subito dopo, esegue un parsing molto semplificato del corpo della richiesta (che si aspetta in formato `name=...`) tramite `sscanf` per estrarre il nome da inserire.

```
int db_create_user(sqlite3* db, const char* name) {
    sqlite3_stmt* stmt;
    const char* sql = "INSERT INTO users (name) VALUES (?);";

    if (sqlite3_prepare_v2(db, sql, -1, &stmt, NULL) !=
   SQLITE_OK) {
        return 0;
    }
    // parsing basico del body: "name=Sandi"
    char real_name[100];
    sscanf(name, "name=%s", real_name);
```

Listing 10: **db_handler.c** - Create (POST /users): preparazione (1/2)

In questa parte finale, il nome estratto (`real_name`) viene collegato (associato) in modo sicuro al segnaposto `?` nella query preparata, utilizzando `sqlite3_bind_text`. Questo è il passaggio che garantisce la protezione da SQL injection. Infine, la query viene eseguita con `sqlite3_step` e la funzione restituisce un valore booleano che indica il successo dell'operazione.

```
    // sssocia il nome alla query
    sqlite3_bind_text(stmt, 1, real_name, -1, SQLITE_STATIC);

    // esegue la query
    int rc = sqlite3_step(stmt);

    // rilascia le risorse
    sqlite3_finalize(stmt);

    return (rc == SQLITE_DONE);
}
```

Listing 11: **db_handler.c** - Create (POST /users): esecuzione (2/2)

3.4 Definizione delle Interfacce e Processo di Compilazione

Il progetto è compilato tramite un `Makefile` che gestisce le dipendenze dei vari moduli e linka le librerie necessarie: `pthread` per il thread pool e `lsqlite3` per il database.

Il `Makefile` automatizza la compilazione. Definisce le variabili per il compilatore (`CC`), i flag (`CFLAGS` per la compilazione, `LDFLAGS` per il linking) e i sorgenti. Il target `all` compila separatamente ogni file `.c` in un file oggetto `.o` e poi li collega tutti insieme (`$^`) per creare l'eseguibile `TARGET`, includendo le librerie `-pthread` e `-lsqlite3`.

```
CC = gcc
CFLAGS = -Wall -Wextra -g -pthread
LDFLAGS = -pthread -lsqlite3

TARGET = rest_server
SRCS = server.c thread_pool.c http_handler.c db_handler.c
OBJS = $(SRCS:.c=.o)

all: $(TARGET)

$(TARGET): $(OBJS)
    $(CC) -o $@ $^ $(LDFLAGS)
    @echo "Build completato: $(TARGET)"

%.o: %.c
    $(CC) $(CFLAGS) -c -o $@ $<

clean:
    rm -f $(OBJS) $(TARGET) rest_api.db

re: clean all

.PHONY: all clean re
```

Listing 12: **Makefile** - Compilazione del progetto

4 Presentazione dei Risultati

A differenza di un server FTP, che si testa con un client `ftp`, un'API REST si testa comunemente con strumenti come `curl` o Postman. Di seguito sono riportati i test eseguiti tramite `curl` dalla riga di comando.

4.1 Compilazione ed Esecuzione

Prima di tutto, si compila il server tramite `make` e lo si avvia. Il server si mette in ascolto sulla porta 8080 (definita in `common.h`).

```
$ make re
rm -f server.o thread_pool.o http_handler.o db_handler.o
   rest_server rest_api.db
gcc -Wall -Wextra -g -pthread -c -o server.o server.c
gcc -Wall -Wextra -g -pthread -c -o thread_pool.o thread_pool.
   c
gcc -Wall -Wextra -g -pthread -c -o http_handler.o
   http_handler.c
gcc -Wall -Wextra -g -pthread -c -o db_handler.o db_handler.c
gcc -o rest_server server.o thread_pool.o http_handler.o
   db_handler.o -pthread -lsqlite3
Build completato: rest_server

$ ./rest_server
Database inizializzato e tabella 'users' pronta.
Thread pool creato con 4 workers.
Server REST in ascolto sulla porta 8080...
```

Listing 13: Compilazione ed avvio del server

4.2 Test dell'API REST con `curl`

In un altro terminale, eseguiamo i comandi `curl` per interagire con l'API.

Creazione di nuovi utenti (POST) Creiamo due utenti. Usiamo `-X POST` per specificare il metodo e `-d` per inviare il body.

```
$ curl -X POST -d "name=Sandi" http://localhost:8080/users
{"status":"created"}

$ curl -X POST -d "name=Mario" http://localhost:8080/users
{"status":"created"}
```

Listing 14: Test: POST /users (Creazione utenti)

Lettura di tutti gli utenti (GET) Verifichiamo che gli utenti siano stati creati.

```
$ curl http://localhost:8080/users
[{"id":1, "name":"Sandi"},{"id":2, "name":"Mario"}]
```

Listing 15: Test: GET /users (Lettura tutti gli utenti)

Lettura di un singolo utente (GET) Recuperiamo l'utente con ID 2.

```
$ curl http://localhost:8080/users/2
{"id":2, "name":"Mario"}
```

Listing 16: Test: GET /users/2 (Lettura singolo utente)

Tentativo di GET su utente non esistente Se richiediamo un ID non valido, il server risponde correttamente con un errore 404.

```
$ curl http://localhost:8080/users/99
{"error":"User not found"}
```

Listing 17: Test: GET /users/99 (Utente non trovato - 404)

Eliminazione di un utente (DELETE) Eliminiamo l'utente con ID 1.

```
$ curl -X DELETE http://localhost:8080/users/1
{"status":"deleted"}
```

Listing 18: Test: DELETE /users/1 (Eliminazione utente)

Verifica finale Controlliamo di nuovo la lista completa per verificare l'avvenuta eliminazione.

```
$ curl http://localhost:8080/users
[{"id":2, "name":"Mario"}]
```

Listing 19: Test: GET /users (Verifica dopo eliminazione)

5 Conclusioni

5.1 Sintesi dei Risultati Ottenuti

Il progetto ha portato alla creazione di un server RESTful concorrente in C pienamente funzionante. Il risultato più significativo è l'implementazione di un'architettura di concorrenza scalabile e moderna, basata su **I/O multiplexing non bloccante (`epoll`) e un thread pool**. Questo design, che separa la gestione delle connessioni dall'elaborazione delle richieste, è alla base di server web ad alte prestazioni come Nginx e Node.js.

Il server è in grado di gestire le operazioni CRUD fondamentali (GET, POST, DELETE) e di garantirne la persistenza attraverso un'integrazione pulita con il database **SQLite**. L'approccio modulare (handler, thread pool, db handler) ha reso il codice gestibile e manutenibile.

5.2 Limitazioni e Criticità del Progetto

Data la natura del progetto, sono state fatte scelte implementative che privilegiano la semplicità concettuale, ma che presentano limitazioni in un contesto reale:

- **Parsing HTTP ingenuo**: Il parser della richiesta, basato su `sscanf`, è estremamente fragile. Non gestisce header HTTP, connessioni keep-alive, chunked transfer-encoding, o body complessi (es. JSON). È sufficiente solo per questo specifico caso d'uso.
- **Parsing del Body limitato**: La funzione `db_create_user` si aspetta un body in formato `name=valore` e lo parsifica con `sscanf`, una pratica non sicura e non estendibile.
- **Generazione JSON manuale**: L'uso di `strcat` e `sprintf` per costruire stringhe JSON è pericoloso (rischio di buffer overflow) e inefficiente.
- **Mancanza di Sicurezza**: Il server non implementa HTTPS (le comunicazioni sono in chiaro) né alcun tipo di autenticazione o autorizzazione.
- **Gestione Connessioni**: Il server adotta un approccio HTTP/1.0, chiudendo la connessione dopo ogni richiesta. Manca il supporto al keep-alive (HTTP/1.1), fondamentale per le performance.

5.3 Proposte per Sviluppi Futuri

Partendo dalla solida base di concorrenza, il progetto potrebbe essere esteso in diverse direzioni per renderlo "production-ready":

- **Integrazione di un Parser HTTP**: Sostituire `sscanf` con una libreria di parsing HTTP robusta e ottimizzata, come `http-parser` di Node.js (scritta in C).

- **Supporto JSON completo**: Integrare una libreria C per la gestione del JSON, come `jansson` o `cJSON`, per parsificare i body delle richieste POST in entrata e generare le risposte in modo sicuro.
- **Supporto HTTPS**: Implementare la crittografia TLS/SSL utilizzando la libreria `OpenSSL` per garantire comunicazioni sicure.
- **Autenticazione**: Aggiungere un livello di autenticazione, ad esempio tramite API key passate in un header (es. `Authorization: Bearer <token>`).
- **Estensione dell'API**: Aggiungere il supporto ai metodi `PUT` o `PATCH` per l'aggiornamento degli utenti, completando l'intero set di operazioni CRUD.
- **Connection pooling per SQLite**: Implementare un pool di connessioni al database per migliorare le performance in scenari ad alto carico.