

LABRESID Relazione Progetto Finale: ShareCare - piattaforma solidale per donazioni e acquisti benefici

Fabiana Presti 554563

1. Problem Statement

L'accesso a strumenti digitali per supportare cause benefiche è spesso vincolato a piattaforme complesse o a costi elevati. Il problema affrontato in questo progetto è la realizzazione di un sistema leggero e personalizzabile, in grado di:

- permettere agli utenti di effettuare donazioni in modo rapido;
- offrire una vetrina di articoli solidali acquistabili;
- gestire tutte le operazioni tramite un server REST minimale;
- funzionare senza autenticazione reale, ma con una UI fittizia;
- garantire la gestione concorrente di più richieste tramite threading.

Il progetto utilizza un server REST: "un'API REST è un'interfaccia di programmazione che usa HTTP per gestire dati remoti" (da "Tutti vogliono fare REST", di Andrea Chiarelli).

In questo caso viene implementato un server REST, senza l'utilizzo di API, interamente in C.

2. Stato dell'Arte

Attualmente esistono molte piattaforme di crowdfunding e donazione (es. GoFundMe, PayPal Donate, Tiltify), ma risultano dispersive e complicate. Realizzare un hub in cui è possibile donare direttamente, ordinare merchandising solidale e condividere la propria esperienza online permette di familiarizzare con tematiche complesse e rendere le donazioni più accessibili.

In ambito accademico, la realizzazione di server REST in linguaggio C è meno comune, ma consente di comprendere il basso livello, la concorrenza, la gestione HTTP e l'interazione con database relazionali.

3. Metodologia ed Implementazione

Il progetto è stato sviluppato seguendo un approccio modulare, con separazione tra:

Backend (C)

- Server multithread che gestisce richieste HTTP su porta 8080.
- Uso di `pthread` per servire ogni client con un thread dedicato.
- Socket TCP per la comunicazione tra browser e server.
- Parsing manuale di metodo, path e body della richiesta HTTP.
- Routing REST per gli endpoint: `/enti`, `/articoli`, `/donazioni`, `/checkout`, `/blog`.

Database (SQLite)

- Persistenza dei dati con SQLite.
- Tabelle principali: `Ente_Benefico`, `Articolo_Solidale`, `Transazione`, `Blog`.

0.0.1 Schema E-R

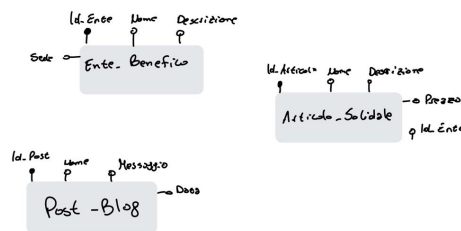


Figura 1: Schema E-R

Frontend (HTML/CSS/JS)

- `index.html`: pagina principale con enti e articoli.
- `paypal.html`: schermata di pagamento simulata.
- `grazie.html`: conferma finale con animazione.
- Integrazione AJAX per chiamate REST e invio dati JSON.
- Blog dinamico con modulo di pubblicazione.

0.1 Gestione del Database: Il Modulo `database.c`

Per la gestione della persistenza dei dati, il progetto ShareCare utilizza un database SQLite locale. L'interazione con il database è incapsulata nel modulo `database.c`, il cui header `database.h` definisce tutte le funzioni esposte.

0.1.1 Inizializzazione e Connessione

La connessione al database viene aperta tramite la funzione:

```
1 int open_db(sqlite3 **db) {  
2     return sqlite3_open("sharecare.db", db);  
3 }
```

Listing 1: Funzione per aprire la connessione

Ogni thread del server apre la propria connessione SQLite, garantendo isolamento delle richieste.

La funzione `init_db()` crea le tabelle solo se non esistono già:

```
1 CREATE TABLE IF NOT EXISTS Ente_Benefico (...);  
2 CREATE TABLE IF NOT EXISTS Articolo_Solidale (...);  
3 CREATE TABLE IF NOT EXISTS Transazione (...);  
4 CREATE TABLE IF NOT EXISTS PostBlog (...);
```

Listing 2: Creazione delle tabelle principali

Quest'ultima tabella `PostBlog` è dedicata al mini-blog per gli utenti.

0.1.2 Inserimento dei Dati

Ogni entità ha la propria funzione di inserimento. Ad esempio, per inserire un ente benefico:

```
1 int insert_ente(sqlite3 *db, const char *nome, const char *descr,  
    const char *sede);
```

La funzione crea un prepared statement, lega i parametri in modo sicuro, ed esegue l'inserimento:

```
1 sqlite3_prepare_v2(db, sql, -1, &stmt, NULL);  
2 sqlite3_bind_text(stmt, 1, nome, -1, SQLITE_TRANSIENT);  
3 ...  
4 sqlite3_step(stmt);  
5 sqlite3_finalize(stmt);
```

Un meccanismo simile viene utilizzato anche per inserire donazioni e post del blog.

0.1.3 Serializzazione in JSON

Per fornire i dati al frontend, le query sui dati vengono convertite in stringhe JSON. La funzione di utilità centrale è:

```
1 static char *json_array(sqlite3 *db, const char *sql);
```

Questa funzione genera un array JSON a partire da una query SQL con la funzione `json_object()` di SQLite:

```
1 SELECT json_object('ID_Ente', ID_Ente, 'Nome', Nome, ...) FROM  
    Ente_Benefico;
```

La funzione gestisce dinamicamente l'allocazione di memoria e concatena ogni elemento JSON in un array complessivo.

Le tre principali funzioni di serializzazione sono:

- `get_enti_json()` – restituisce tutti gli enti benefici
- `get_articoli_json()` – restituisce gli articoli solidali
- `get_blog_json()` – restituisce i post del blog condivisi dagli utenti

0.1.4 Post del Blog

Il blog permette agli utenti di condividere pensieri o esperienze. I dati vengono salvati con:

```
1 int insert_post(sqlite3 *db, const char *nome, const char *msg);
```

Questa funzione esegue un semplice inserimento nella tabella `PostBlog`.

La funzione `get_blog_json()` permette di serializzare i post esistenti per visualizzarli in frontend.

0.2 Interfaccia del modulo database – database.h

Il file `database.h` dichiara le funzioni principali per la gestione del database SQLite utilizzato dal server ShareCare. Sono inclusi i prototipi per l'apertura/chiusura del database, la creazione delle tabelle, l'estrazione dei dati in formato JSON, e l'inserimento di nuovi record.

```
1 #ifndef SHARECARE_DATABASE_H
2 #define SHARECARE_DATABASE_H
3
4 #include <sqlite3.h>
5
6 int open_db(sqlite3 **db);
7 void close_db(sqlite3 *db);
8 int init_db(sqlite3 *db);
9
10 //serializza in JSON
11 char *get_enti_json(sqlite3 *db);
12 char *get_articoli_json(sqlite3 *db);
13
14 //blog
15 char *get_blog_json(sqlite3 *db);
16
17 //inserimenti
18 int insert_ente(sqlite3 *db, const char *nome, const char *descr,
19               const char *sede);
20 int insert_donazione(sqlite3 *db, int id_utente, int id_ente,
21                   double importo);
22 int insert_post(sqlite3 *db, const char *nome, const char *msg)
23     ;
24
25 #endif // SHARECARE_DATABASE_H
```

Listing 3: Prototipi di funzioni in database.h

0.3 Server

Il cuore del sistema **ShareCare** è un server RESTful realizzato in linguaggio C, progettato per gestire simultaneamente molteplici richieste client attraverso l'utilizzo di thread POSIX (pthread). Il file principale include l'inizializzazione del server, la logica di routing per le richieste HTTP e la gestione dei dati attraverso il modulo `database.h`.

Il server è configurato per ascoltare sulla porta 8080:

```
1 #define PORTA 8080
2 #define BUF_SIZE 8192
3 atomic_int contatore_client = 0;
```

L'uso di `atomic_int` permette di contare in maniera sicura e concorrentiale i client connessi.

Inizializzazione del server: nella funzione `main()`, viene aperto il database SQLite e avviato un ciclo infinito che accetta connessioni da nuovi client. Ogni client viene gestito da un nuovo thread:

```
1 while (1) {
2     int client_sock = accept(...);
3     pthread_t tid;
4     struct ThreadArgs *args = malloc(sizeof(struct ThreadArgs));
5     args->sock = client_sock;
6     pthread_create(&tid, NULL, thread_client, args);
7     pthread_detach(tid);
8 }
```

Gestione del client: la funzione `thread_client()` si occupa di aprire una connessione al database (ogni thread ha la sua), elaborare la richiesta ricevuta e chiudere la connessione:

```
1 void *thread_client(void *arg) {
2     struct ThreadArgs *args = (struct ThreadArgs *)arg;
3     int client_sock = args->sock;
4     free(args);
5
6     sqlite3 *db;
7     if (open_db(&db)) { ... }
8     gestisci_client(client_sock, db);
9     close_db(db);
10    close(client_sock);
11    pthread_exit(NULL);
12 }
```

Funzione `gestisci_client`: questa funzione riceve la richiesta HTTP, ne analizza il metodo (GET/POST) e il path, quindi inoltra la gestione a uno dei router:

- GET /enti → `route_get_enti`
- POST /enti → `route_post_enti`
- GET /articoli → `route_get_articoli`
- POST /donazioni → `route_post_donazioni`

- POST /checkout → route_post_checkout
- GET /blog → route_get_blog
- POST /blog → route_post_blog

Ogni route si connette al database per eseguire query e generare risposte JSON, che vengono poi inviate al client con `invia_risposta()`.

File statici: se la richiesta è per una risorsa statica come HTML, CSS, PNG, ecc., viene gestita da `serve_static_file()`, che restituisce il contenuto del file richiesto con il giusto header HTTP.

Infine, il codice supporta anche un semplice blog condiviso attraverso una tabella `PostBlog`, dove gli utenti possono inviare messaggi (POST) e recuperarli (GET).

Questo approccio multithreaded consente un'elevata reattività e scalabilità del server in ambienti a bassa latenza come quello locale.

0.4 Interfaccia principale: index.html

La homepage di *ShareCare* è progettata in HTML con elementi dinamici controllati da JavaScript. Essa presenta tre sezioni principali: uno slider informativo, l'elenco degli enti benefici e gli articoli solidali acquistabili. Inoltre, include l'accesso alla pagina del blog.

Header e navigazione

```

1 <header>
2   <h1>ShareCare</h1>
3   <p>Condividi. Aiuta. Dona.</p>
4   <nav>
5     <a href="#enti">Enti benefici</a>
6     <a href="#articoli">Articoli solidali</a>
7     <a href="blog.html">Blog</a>
8   </nav>
9 </header>

```

Listing 4: Header con titolo e navigazione

Questa sezione offre la navigazione principale all'interno della singola pagina o verso la sezione blog esterna.

Slider tematico automatico

```

1 <section class="slider-cause">
2   <div class="cause-list" id="cause-list">
3     <div class="cause">
4       
5       <h3>Sostegno all'infanzia con UNICEF</h3>
6       <p>Garantisci istruzione, acqua potabile e cure sanitarie
          ai bambini nel mondo.</p>
7     </div>
8     <!-- ... -->
9   </div>

```

```
10 </section>
```

Listing 5: Contenuto dello slider informativo

```
1 let index = 0;
2 const causes = document.querySelectorAll(".cause");
3 const causeList = document.getElementById("cause-list");
4
5 setInterval(() => {
6   index = (index + 1) % causes.length;
7   causeList.style.transform = `translateX(-${index * 100}vw)`;
8 }, 10000);
```

Listing 6: Scorrimento automatico dello slider

Il codice JavaScript realizza lo scorrimento ogni 10 secondi.

Caricamento dinamico degli enti benefici

```
1 fetch("http://localhost:8080/enti")
2   .then(response => response.json())
3   .then(dati => {
4     const ul = document.getElementById("lista-enti");
5     ul.innerHTML = "";
6     dati.forEach(ente => {
7       const li = document.createElement("li");
8       li.innerHTML = `
9         <div class="ente-box">
10           
12           <div>
13             <strong>${ente.Nome}</strong><br>
14             <em>${ente.Descrizione || "Nessuna descrizione"}</em>
15             <br>
16             <small>${ente.Sede || ""}</small><br>
17             <button class="btn-dona" data-ente="${ente.ID_Ente}"
18               data-importo="10">
19               Dona con PayPal
20             </button>
21           </div>
22         </div>`;
23       ul.appendChild(li);
24     });
25   });
```

Listing 7: Fetch dinamico degli enti

Questo frammento consente di recuperare e visualizzare dinamicamente gli enti dal server.

Visualizzazione articoli solidali

```

1 fetch("http://localhost:8080/articoli")
2   .then(r => r.ok ? r.json() : Promise.reject(r))
3   .then(articoli => {
4     const container = document.querySelector("#articoli-container");
5     container.innerHTML = "";
6     articoli.forEach(a => {
7       const prezzo = parseFloat(a.Prezzo || 0);
8       const card = document.createElement("div");
9       card.className = "card";
10      card.innerHTML = `
11        
12        <h3>${a.Nome}</h3>
13        <p>${a.Descrizione || ""}</p>
14        <p class="prezzo">    ${prezzo.toFixed(2)}</p>
15        <button class="btn-acquista" data-articolo="${a.
16          ID_Articolo}"
17          data-ente="${a.ID_Ente}" data-importo="${prezzo.
18            toFixed(2)}">
19          Acquista con PayPal
20        </button>`;
21      container.appendChild(card);
22    });
23  });

```

Listing 8: Render dinamico articoli solidali

Integrazione con la pagina paypal.html

```

1 document.body.addEventListener('click', e => {
2   if (e.target.classList.contains('btn-dona')) {
3     const b = e.target;
4     const ente = b.dataset.ente;
5     const importo = b.dataset.importo;
6     location.href = `paypal.html?tipo=donazione&id_ente=${ente}&
7       importo=${importo}`;
8   }
9   if (e.target.classList.contains('btn-acquista')) {
10    const b = e.target;
11    const ente = b.dataset.ente;
12    const art = b.dataset.articolo;
13    const imp = b.dataset.importo;
14    location.href = `paypal.html?tipo=acquisto&id_ente=${ente}&
15      id_articolo=${art}&importo=${imp}`;
16  }
17 });

```

Listing 9: Gestione click su bottoni PayPal

Questo codice permette di reindirizzare l'utente alla schermata di pagamento simulata con i parametri URL.

Uso dei <template> HTML

Infine, sono inclusi due template HTML per clonare dinamicamente le card:

```
1 <template id="ente-template">
2   <li class="card">
3     <h3 class="nome"></h3>
4     <p class="descrizione"></p>
5     <small class="sede"></small>
6   </li>
7 </template>
8
9 <template id="articolo-template">
10  <div class="card">
11    <img class="foto" />
12    <h3 class="nome"></h3>
13    <p class="descrizione"></p>
14    <p class="prezzo"></p>
15  </div>
16 </template>
```

Listing 10: Template per enti e articoli

0.5 Pagina Blog: blog.html

La pagina blog.html consente agli utenti di **condividere la propria esperienza solidale**, tramite un'interfaccia semplice che interagisce dinamicamente con il backend. I post vengono salvati nel database e caricati in tempo reale all'apertura o dopo un nuovo inserimento.

Struttura HTML

La pagina è composta da due elementi principali: il form di inserimento e l'area che mostra i post salvati.

```
1 <form id="form-blog">
2   <input type="text" id="nome-blog" placeholder="Il tuo nome"
3     required />
4   <textarea id="messaggio-blog" placeholder="Scrivi qui la tua
5     esperienza..." required></textarea>
6   <button type="submit">Pubblica</button>
7 </form>
```

Listing 11: Form per l'invio di un post

Il contenitore per i post esistenti viene inizialmente popolato con un messaggio di caricamento:

```
1 <div id="post-container">
2   <p>Caricamento post...</p>
```

```
3 </div>
```

Listing 12: Contenitore dinamico dei post

Caricamento automatico dei post dal server

Una funzione JavaScript esegue una GET verso l'endpoint /blog e inserisce dinamicamente i post nella pagina.

```
1 function caricaPost() {
2   fetch("http://localhost:8080/blog")
3     .then(res => res.json())
4     .then(posts => {
5       const container = document.getElementById("post-container")
6       ;
7       container.innerHTML = "";
8       posts.forEach(p => {
9         const div = document.createElement("div");
10        div.className = "post";
11        div.innerHTML = `<strong>${p.Nome}</strong> (${p.Data})<br>${p.Messaggio}`;
12        container.appendChild(div);
13      });
14    });
15 }
```

Listing 13: Funzione di caricamento dei post

Questa funzione viene invocata anche all'avvio della pagina per visualizzare i post esistenti:

```
1 caricaPost();
```

Invio di un nuovo post con fetch POST

Alla pressione del pulsante "Pubblica", il form viene intercettato con JavaScript. I dati vengono inviati al server in formato JSON.

```
1 document.getElementById("form-blog").addEventListener("submit", e
2   => {
3     e.preventDefault();
4     const nome = document.getElementById("nome-blog").value;
5     const msg = document.getElementById("messaggio-blog").value;
6
7     fetch("http://localhost:8080/blog", {
8       method: "POST",
9       headers: { "Content-Type": "application/json" },
10      body: JSON.stringify({ Nome: nome, Messaggio: msg })
11    })
12    .then(res => {
13      if (res.ok) {
14        caricaPost();
15        // aggiorna la lista post
16      }
17    });
18  });
```

```

14     document.getElementById("form-blog").reset(); // reset
        form
15 } else {
16     alert("Errore nel salvataggio");
17 }
18 });
19 });

```

Listing 14: Invio nuovo post con fetch POST

0.6 Pagina di Checkout: paypal.html

La pagina `paypal.html` simula un'interfaccia PayPal semplificata, utilizzata per confermare una donazione o un acquisto solidale.

Struttura del form

Il form HTML consente l'inserimento delle credenziali (email e password) ed eventualmente dell'indirizzo di spedizione (solo per acquisti).

```

1 <form id="paypal-form">
2   <label>Email PayPal:<br>
3     <input type="email" required style="width:100%; padding:8px;"
        />
4   </label><br><br>
5
6   <label>Password:<br>
7     <input type="password" required style="width:100%; padding:8
        px;" />
8   </label><br><br>
9
10  <div id="indirizzo-box" style="display:none;">
11    <label>Indirizzo di spedizione:<br>
12      <input type="text" name="indirizzo" style="width:100%;
        padding:8px;" />
13    </label><br><br>
14  </div>
15
16  <button type="submit" style="width:100%; background:#ffc439;
        border:none; padding:10px; font-weight:bold;">
17    Procedi al pagamento
18  </button>
19 </form>

```

Listing 15: Form simulato PayPal

Parametri dalla query string e comportamento dinamico

Lo script JavaScript legge i parametri dalla `query string` per capire se l'utente sta effettuando una **donazione** o un **acquisto**. In base al tipo, modifica il messaggio visualizzato e mostra l'input per l'indirizzo.

```

1 const params = new URLSearchParams(window.location.search);
2 const tipo = params.get("tipo");
3 const importo = params.get("importo");
4 const id_ente = params.get("id_ente");
5 const id_articolo = params.get("id_articolo");
6
7 const info = document.getElementById("info");
8 const indirizzoBox = document.getElementById("indirizzo-box");
9
10 if (tipo === "donazione") {
11     info.innerHTML = 'Stai per donare <strong>    ${importo}</strong> all\'ente <strong>ID #${id_ente}</strong>';
12 } else if (tipo === "acquisto") {
13     info.innerHTML = 'Stai per acquistare un articolo solidale da <strong>ID Ente #${id_ente}</strong> per <strong>    ${importo}</strong>';
14     indirizzoBox.style.display = "block";
15 } else {
16     info.innerHTML = "Errore: tipo di operazione sconosciuto.";
17 }

```

Listing 16: Gestione parametri URL e messaggio

Gestione invio del form

Il submit del form è intercettato via JavaScript. Successivamente si viene reindirizzati alla pagina di ringraziamento.

```

1 document.getElementById("paypal-form").addEventListener("submit",
2     e => {
3     e.preventDefault();
4     window.location.href = "grazie.html";
5 });

```

Listing 17: Gestione invio simulato del pagamento

4. Risultati

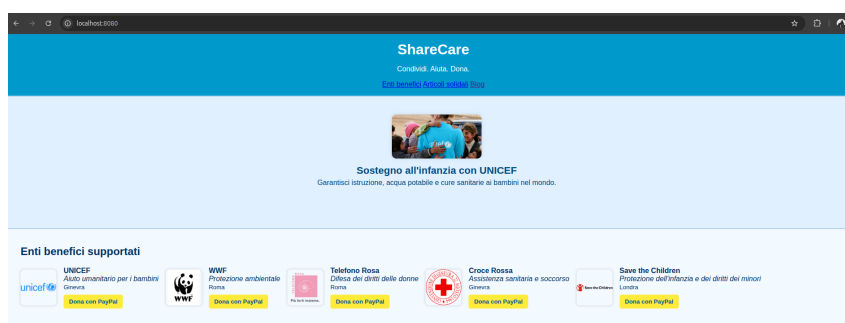
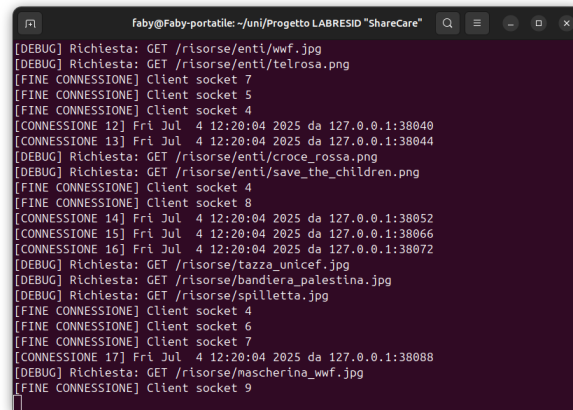


Figura 2: Schermata Home

La schermata home. Contiene tutti gli enti che accettano donazioni dirette. Un banner scorre ogni 10 secondi, mostrando cause benefiche diverse.



```
faby@Faby-portable: ~/uni/Progetto LABRESID "ShareCare"
[DEBUG] Richiesta: GET /risorse/enti/wwf.jpg
[DEBUG] Richiesta: GET /risorse/enti/telrosa.png
[FINE CONNESSIONE] Client socket 7
[FINE CONNESSIONE] Client socket 5
[FINE CONNESSIONE] Client socket 4
[CONNESSIONE 12] Fri Jul 4 12:20:04 2025 da 127.0.0.1:38040
[CONNESSIONE 13] Fri Jul 4 12:20:04 2025 da 127.0.0.1:38044
[DEBUG] Richiesta: GET /risorse/enti/croce_rossa.png
[DEBUG] Richiesta: GET /risorse/enti/save_the_children.png
[FINE CONNESSIONE] Client socket 4
[FINE CONNESSIONE] Client socket 8
[CONNESSIONE 14] Fri Jul 4 12:20:04 2025 da 127.0.0.1:38052
[CONNESSIONE 15] Fri Jul 4 12:20:04 2025 da 127.0.0.1:38066
[CONNESSIONE 16] Fri Jul 4 12:20:04 2025 da 127.0.0.1:38072
[DEBUG] Richiesta: GET /risorse/tazza_unicef.jpg
[DEBUG] Richiesta: GET /risorse/bandiera_palestina.jpg
[DEBUG] Richiesta: GET /risorse/spilletta.jpg
[FINE CONNESSIONE] Client socket 4
[FINE CONNESSIONE] Client socket 6
[FINE CONNESSIONE] Client socket 7
[CONNESSIONE 17] Fri Jul 4 12:20:04 2025 da 127.0.0.1:38088
[DEBUG] Richiesta: GET /risorse/mascherina_wwf.jpg
[FINE CONNESSIONE] Client socket 9
```

Figura 3: Debug da terminale

Il terminale mostra le connessioni nuove e terminate insieme alle richieste.



Email PayPal:

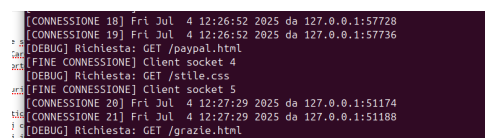
Password:

Indirizzo di spedizione:

Procedi al pagamento

Figura 4: Simulazione di un acquisto

Per simulare un acquisto si viene reindirizzati alla schermata di pagamento. Se si tratta di un articolo viene chiesto anche l'indirizzo di spedizione, altrimenti solo le credenziali.



```
[CONNESSIONE 18] Fri Jul 4 12:26:52 2025 da 127.0.0.1:57728
[CONNESSIONE 19] Fri Jul 4 12:26:52 2025 da 127.0.0.1:57736
[DEBUG] Richiesta: GET /paypal.html
[FINE CONNESSIONE] Client socket 4
[DEBUG] Richiesta: GET /stile.css
[FINE CONNESSIONE] Client socket 5
[CONNESSIONE 20] Fri Jul 4 12:27:29 2025 da 127.0.0.1:51174
[CONNESSIONE 21] Fri Jul 4 12:27:29 2025 da 127.0.0.1:51188
[DEBUG] Richiesta: GET /grazie.html
[FINE CONNESSIONE] Client socket 7
```

Figura 5: Debug durante la transazione

Nel terminale è possibile osservare il funzionamento del server REST, che riceve e gestisce le connessioni HTTP in maniera concorrente. Ogni richiesta viene tracciata con messaggi di debug, utili per il monitoraggio e la diagnostica.

- Ogni connessione viene identificata da un numero progressivo: ad esempio CONNESSIONE 18, CONNESSIONE 19, ecc.
- È indicato il timestamp della connessione (es. Fri Jul 4 12:26:52 2025) e l'indirizzo IP del client (in questo caso 127.0.0.1, ossia localhost).
- Viene mostrata la risorsa richiesta dal client tramite il metodo GET, ad esempio:

- /paypal.html
 - /stile.css
 - /grazie.html
- Il messaggio `FINE CONNESSIONE Client socket N` segnala l'avvenuta chiusura del socket lato server dopo l'elaborazione della richiesta.

Questi log dimostrano che il server è in grado di:

- Servire file statici richiesti dai client
- Identificare ogni connessione entrante
- Gestire la chiusura dei socket



Figura 6: Blog

Il blog permette agli utenti di condividere le loro esperienze online e scambiare opinioni.

```
[CONNESSIONE 24] Fri Jul 4 12:37:11 2025 da 127.0.0.1:49164
[DEBUG] Richiesta: GET /blog
[DEBUG] Risposta inviata: 200 OK
[FINE CONNESSIONE] Client socket 5
[CONNESSIONE 25] Fri Jul 4 12:39:47 2025 da 127.0.0.1:39362
[DEBUG] Richiesta: POST /blog
[DEBUG] Risposta inviata: 201 Created
[FINE CONNESSIONE] Client socket 4
[CONNESSIONE 26] Fri Jul 4 12:39:47 2025 da 127.0.0.1:39364
[DEBUG] Richiesta: GET /blog
[DEBUG] Risposta inviata: 200 OK
[FINE CONNESSIONE] Client socket 5
```

Figura 7: Debug durante scrittura su Blog

Il terminale mostra un ciclo completo di interazione REST con l'endpoint `/blog`. Dopo una GET iniziale per recuperare i post, un utente invia un nuovo contributo tramite POST. Il server risponde con codice 201 Created, confermando l'avvenuta scrittura nel database. Una successiva GET dimostra che i dati sono persistenti e correttamente disponibili, riflettendo il comportamento atteso di un'API REST conforme.

5. Conclusione e sviluppi futuri

Il progetto **ShareCare** dimostra come sia possibile realizzare una piattaforma REST completa in linguaggio C, con supporto multithread, persistenza dati e interfaccia web leggera e navigabile.

Sviluppi futuri

- Aggiunta di autenticazione vera (es. JWT o sessioni).
- Dashboard per enti con statistiche delle donazioni.
- Upload dinamico di immagini e articoli.
- Protezione del blog da spam o contenuti non validi.