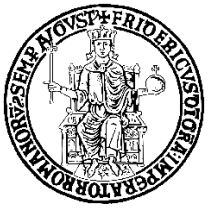
UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II



CORSO DI LABORATORIO DI SISTEMI OPERATIVI

ANNO ACCADEMICO 2022/2023

GALLERYINSIGHTS

Candidati

Pasquale Orlando – N86003266

Alessia Verrazzo – N86003326

Indice

1 Introduzione

GalleryInsights è un sistema **client-server** che consente ai visitatori di un museo di ottenere informazioni circa le esibizioni e le opere esposte.

1.1 Funzionalità offerte

Il sistema è progettato per aiutare gli utenti ad ottenere le informazioni necessarie per godersi a pieno la loro visita in un museo e offre, quindi, una serie di funzionalità per soddisfarne le esigenze.

- I visitatori possono **registrarsi** al museo con e-mail e password
- I visitatori possono effettuare il **login** in qualsiasi momento con le loro credenziali
- **Ogni visitatore appartiene ad una o più categorie**, specificate in fase di accesso e modificabili in ogni momento, in modo da personalizzare a pieno la sua esperienza

1.2 Tecnologie utilizzate

Il sistema è stato sviluppato utilizzando un insieme di tecnologie per garantire un'esperienza utente eccellente e prestazioni ottimali.

1.2.1 Linguaggi di programmazione

- **C**: utilizzato come linguaggio principale per lo sviluppo del backend dell'applicativo
- **Java**: utilizzato come linguaggio principale dell'applicativo Android per tablet

1.2.2 Database

- **MySQL**: DBMS interrogato dal server in cui sono memorizzati gli utenti, le mostre e le opere del museo

1.2.3 Containerizzazione

- **Docker**: utilizzato per la creazione dei container del server del database

2 Guida alla compilazione e all'uso

2.1 Compilazione ed installazione

Per rendere l'installazione più semplice abbiamo utilizzato Docker e docker-compose.

NOTA BENE: per eseguire i container è necessario installare il Docker Engine. La guida su come farlo è disponibile al seguente link (Install using apt repository). Prima di eseguire docker-compose bisogna far partire il Docker Engine attraverso il comando:

```
sudo dockerd
```

Il server è strutturato in diverse directory, ognuna delle quali contiene diversi file .c e .h rappresentanti delle funzionalità (ad esempio logging o accesso al database). Per compilarlo ed eseguirlo abbiamo definito un Dockerfile

```
# base image
FROM gcc:latest
# copy source files
COPY . /server
# enter in the copied directory
WORKDIR /server/
# compile c program
RUN gcc -o server $(mysql_config --cflags) server.c database/database.c
database/repository/user_repository.c
database/repository/piece_repository.c
database/repository/exhibition_repository.c
database/service/user_service.c database/service/piece_service.c
database/service/exhibition_service.c utils/utils.c utils/log.c
$(mysql_config --libs)
# execute the server
CMD ["./server"]
```

che prende l'immagine di gcc, copia nel container i file sorgente, li compila ed infine esegue.

Per quanto riguarda il database MySQL, questo viene eseguito all'interno di un altro container come definito all'interno del file docker-compose.yml:

```
version: '3.8'
services:
  mysql_db:
    image: mysql:latest
    restart: always
    command: --default-authentication-plugin=caching_sha2_password
    environment:
      MYSQL_DATABASE: GalleryInsights
      MYSQL_USER: mysql
      MYSQL_PASSWORD: mysql
      MYSQL_ROOT_PASSWORD: root
    healthcheck:
      test: ["CMD", 'mysqladmin', 'ping', '-h', 'localhost', '-u',
'root', '-p$$MYSQL_ROOT_PASSWORD' ]
      timeout: 20s
      retries: 10
    ports:
      - "3306:3306"
```

```
    container_name: mysql_db
    volumes:
      - db-data:/var/lib/mysql

  server:
    build: .
    container_name: server
    restart: always
    environment:
      DB: mysql_db
    ports:
      - "8888:8888"
    depends_on:
      mysql_db:
        condition: service_healthy
    links:
      - mysql_db

volumes:
  db-data:
    driver: local
```

Questo file Docker Compose configura due servizi: un database MySQL (mysql_db) e un server (server). Il server dipende dal servizio MySQL e comunica con esso attraverso la variabile d'ambiente "DB" (abbiamo utilizzato una variabile d'ambiente di docker in modo da poter utilizzare la funzione getenv nel codice sorgente del server). Entrambi i servizi sono configurati per essere riavviati automaticamente (restart: always). Nello specifico:

- **mysql_db**: utilizza l'immagine MySQL più recente, imposta le variabili d'ambiente per la configurazione del database e definisce un healthcheck per verificare che il server sia partito. Viene mappata la porta 3306 del container a quella del sistema host e viene utilizzato un volume chiamato db-data per la persistenza dei dati del database.
- **server**: costruisce l'immagine del server dall'attuale contesto della build, imposta variabili d'ambiente per la connessione al database MySQL e dipende dal servizio mysql_db con una condizione di dipendenza basata sulla salute del servizio. Inoltre viene mappata la porta 8888 del container a quella del sistema host.

Quindi eseguendo il comando

```
docker-compose build
```

all'interno della directory dove è presente il file docker-compose.yml, verranno scaricate le immagini e creati i container che eseguiranno il server ed il database.

2.2 Esecuzione

Per eseguire server e database basta il comando:

```
docker-compose up
```

all'interno della directory del file docker-compose.yml. Il server viene eseguito senza nessuna personalizzazione. Altrimenti si può procedere con una piccola configurazione del server modificando l'ultima riga del Dockerfile con:

```
./server <-l | --log | --log=true | --log=false> <-i | --insert | --
insert=true | --insert=false>
```

dove i parametri riguardano due funzionalità del server:

1. **Logging su file**: con parametro -l oppure --log oppure --log=true, i log vengono scritti su file di testo, altrimenti sullo standard error. Viene creato un file di testo, se non esiste, su cui scrive fino a quando non viene spento. In caso di riavvio nella stessa data, viene selezionato il file creato in precedenza.
2. **Inserimento dati mock**: con parametro -i oppure --insert oppure --insert=true, all'avvio del server vengono inseriti dei dati all'interno del database riguardanti opere ed esibizioni. Non viene creato nessun utente.

Una volta effettuata la modifica basta eseguire questi due comandi:

```
docker-compose build
docker-compose up
```

2.3 Esecuzione con Valgrind

È possibile eseguire il server con **Valgrind**, uno strumento molto potente che ci permette di vedere se sono presenti **memory leaks** nel codice del server. Eventuali memory leaks potrebbero portare a crash e/o a comportamenti non prevedibili. Per utilizzare Valgrind basta eseguire il comando dopo aver compilato:

```
valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes

--verbose --log-file=valgrind-out.txt ./server
```

3 Protocollo di comunicazione

In questa sezione vediamo come è stata implementata la comunicazione tra client e server e il formato utilizzato per lo scambio delle informazioni.

Il software utilizza il protocollo di comunicazione **TCP** per facilitare lo scambio di dati tra i dispositivi connessi in una rete. Il protocollo TCP è stato scelto per la sua **affidabilità** ed **efficienza** e inoltre implementa meccanismi di **gestione degli errori** avanzati per garantire che i dati siano correttamente ricevuti anche in condizioni di rete instabili o con perdite di pacchetti.

Si è preferito fare in modo che tra i client e il server non ci fosse una connessione perenne; la connessione viene chiusa ogni qual volta la richiesta viene soddisfatta.

I messaggi che il **client invia al server** seguono questo formato:

```
funzionalità parametro1:parametro2[[:parametroN]…]
```

Se la funzionalità richiesta non esiste, oppure per quella determinata funzionalità non ha inviato parametri a sufficienza, il server invia un messaggio di errore.

I messaggi che il **server invia al client**, in caso di più elementi, seguono il formato:

```
START
attributo1:attributo2[[:attributoN]…]
attributo1:attributo2[[:attributoN]…]
attributo1:attributo2[[:attributoN]…]
…
END
```

in modo tale che il client possa capire quando deve smettere di memorizzare le informazioni ricevute.

4 Dettagli implementativi

Approfondiamo quindi quelli che sono i dettagli implementativi del server e del client.

4.1 Server

Il server è sviluppato in linguaggio C. Innanzitutto vengono inizializzati il logger e la connessione al database. Successivamente viene creata la socket con AF_INET e INADDR_ANY per permettere la connessione al server da qualsiasi indirizzo; viene effettuato il **bind** e ci si mette in ascolto dei client con **listen**.

Per rendere il server concorrente si è deciso di utilizzare la funzione **select**. La funzione **select**() in linguaggio C è utilizzata per gestire più socket in un ambiente **multiclient**, consentendo al server di monitorare più connessioni contemporaneamente senza la necessità di utilizzare thread separati (rischiando attacchi DDOS) per ogni client. Vediamone quindi i vantaggi:

- **Efficienza**: è una chiamata di sistema efficiente che permette al server di attendere su più socket in modo non bloccante. In questo modo, il server può gestire più connessioni senza la necessità di creare un thread separato per ciascuna
- **I/O non bloccante**: consente di eseguire operazioni di I/O non bloccanti. Ciò significa che il server può attendere su più socket contemporaneamente senza essere bloccato da una singola operazione di lettura o scrittura
- **Rilevamento di eventi**: segnala al server quando uno o più socket sono pronti per la lettura o la scrittura. Questo permette al server di concentrare le risorse solo sui socket che richiedono un'azione
- **Gestione di timeout**: consente di specificare un timeout, il che significa che il server può attendere per un certo periodo di tempo prima di controllare nuovamente i socket. Questo è utile per implementare logiche di timeout
- **Minore complessità rispetto ai thread**: evita la complessità aggiuntiva associata alla sincronizzazione e alla gestione dei thread, che può essere difficile e soggetta a problemi di concorrenza
- **Facilità di utilizzo**: Non richiede una grande complessità di gestione rispetto ad altre tecniche come l'uso di thread. Se il numero di client è relativamente basso e non richiede complessità aggiuntiva, select() può essere la scelta più semplice

Tuttavia, è importante notare che select ha alcune limitazioni, come la gestione di un numero massimo di socket (spesso limitato a un valore come **FD_SETSIZE**, che può variare a seconda del sistema). Inoltre, il codice utilizzando select può diventare complesso e difficile da leggere quando si gestiscono molti socket.

Quando su un descrittore di socket arriva un'operazione di I/O, il messaggio del client (letto con **read**) e il descrittore della socket vengono inviati a una funzione che fa una prima scrematura sulla funzionalità richiesta dall'utente. Successivamente viene chiamata la funzione vera e propria relativa alla funzionalità scelta che, in primis "spacchetta" il messaggio estraendone i parametri, poi effettua i calcoli/query sul database e infine invia tramite una **write** il messaggio sulla socket.

4.1.1 Logging

Il server è in grado di effettuare il logging delle operazioni sia su standard error che su file di testo. Ogni riga che viene stampata contiene informazioni su: data e ora, file, riga, tipo di log, messaggio. La funzione principale **logger_log è variadica**, vale a dire che accetta un numero non prefissato di parametri (funziona come printf). Abbiamo inserito **cinque livelli di log**: OFF, DEBUG, INFO, WARNING, ERROR. Il logger viene inizializzato con il livello **MINIMO** da stampare. Ad esempio, se viene inizializzato con WARNING, allora non verranno mostrati i messaggi di DEBUG e di INFO.

Se l'utente decide di stampare i log su un file di testo, questo viene creato se non esiste e rinominato con la data odierna. Nel caso in cui il server rimanga in esecuzione per più giorni, non verranno creati altri file di testo allo scattare della mezzanotte, ma solo se il server viene riavviato.

4.1.2 Database

Come accennato nell'introduzione, abbiamo utilizzato il noto DBMS **MySQL**. Nel codice sorgente del server abbiamo diversi file per accedere al database e per effettuare diverse operazioni in base alle informazioni che ci interessano. Abbiamo diviso le funzionalità in **due directory**:

- **Service**: file che contengono tutte le funzioni che a partire dal messaggio del client, estraggono i parametri, li controllano e se tutto va a buon fine chiamano le funzioni che accedono ai dati contenuti nel database e restituiscono la risposta al client
- **Repository**: file che contengono tutte le funzioni che accedono al database, ovvero effettuano le query e memorizzano le informazioni in strutture dati apposite

4.2 Client

Il client Android del nostro software è stato sviluppato utilizzando il linguaggio di programmazione Java, noto per la sua portabilità e affidabilità. Abbiamo adottato il **pattern architetturale Model-View-Presenter (MVP)** per garantire una separazione chiara delle responsabilità e una gestione efficace dello stato dell'applicazione.

- **Model**: nel contesto dell'applicazione Android, il Model rappresenta la parte responsabile della gestione dei dati e della logica di business. Abbiamo strutturato il Model per gestire il recupero e la manipolazione dei dati provenienti dal server, nonché la loro persistenza in locale
- **View**: è la componente dell'applicazione Android responsabile dell'interfaccia utente e dell'interazione con l'utente. Abbiamo progettato la View in modo modulare, garantendo una separazione netta tra l'interfaccia utente e la logica di presentazione
- **Presenter**: funge da intermediario tra il Model e la View. È responsabile della gestione degli eventi dell'utente e dell'aggiornamento della View in base alle modifiche apportate al Model. Inoltre, il Presenter coordina la comunicazione tra la View e il Model, garantendo una corretta sincronizzazione delle informazioni

L'adozione del pattern MVP ha portato notevoli vantaggi al nostro sviluppo. Ha favorito una separazione pulita delle responsabilità, consentendoci di concentrarci su singoli aspetti dell'applicazione senza sovraccaricare una singola componente. Inoltre, la struttura MVP ha reso il codice più **manutenibile** e **testabile**, agevolando l'identificazione e la correzione di eventuali bug o problemi di performance.

Per evitare crash e freeze dell'applicativo, dovuti al sovraccarico del thread principale, **tutte le operazioni che non riguardano l'interfaccia grafica vengono eseguite in thread anonimi**. Le operazioni di aggiornamento della UI vengono eseguiti grazie al metodo **runOnUiThread**.

Per la visualizzazione di liste di elementi abbiamo utilizzato classi **Adapter**. Gli adapter in Android Studio sono componenti essenziali per collegare dati a componenti dell'interfaccia utente, come ListView, RecyclerView e Spinner. Servono a fornire un'interfaccia tra i dati e la visualizzazione dei dati stessi. Gli adapter sono particolarmente utili quando si tratta di visualizzare elenchi o elenchi di dati complessi all'interno di un'app Android.

Oltre ai package Model, View e Presenter abbiamo definito altri due package: uno dedicato agli **Adapter**, e l'altro chiamato **Service**; quest'ultimo contiene tutte le classi e i metodi che servono ad ottenere/inviare dati dal/al server.

5 Test effettuati

Per testare il codice del server abbiamo creato un programma in linguaggio C che simula quelle che sono le richieste di più client. Il programma crea N processi (dove N viene dato in input da linea di comando) i quali richiedono una delle funzionalità messe a disposizione del server. Questo ci permette di verificare non solo gli output attesi dalle diverse funzioni ma anche di testare come si composta il server con più client diversi.

Il codice test sviluppato è il seguente:

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <arpa/inet.h>
#include <wait.h>
#include <time.h>
#include <string.h>

#define MAX_BUFFER_READ 100
#define BUFSIZE 68000
#define PORT 8888

typedef struct random_user {
      char* name;
      char* email;
      char* password;
} random_user_t;

random_user_t users[] = {
    {"AliceSmith", "alice.smith@example.com", "password123"},
    {"BobJohnson", "bob.johnson@example.com", "securePassword"},
    {"CharlieBrown", "charlie.brown@example.com", "ilovepeanuts"},
    {"DavidLee", "david.lee@example.com", "davidpassword"},
    {"Ella Davis", "ella.davis@example.com", "password12345"},
    {"FrankWilson", "frank.wilson@example.com", "frankspassword"},
    {"GraceTaylor", "grace.taylor@example.com", "gracefulPassword"},
    {"HenryMoore", "henry.moore@example.com", "henryssecret"},
    {"IsabellaClark", "isabella.clark@example.com", "isabellapassword"},
    {"Jack Thomas", "jack.thomas@example.com", "jackpassword"}
};

//Change setup to send whetever you want
void func(int sockfd, int action){

      char buffWrite[BUFSIZE + 1];
      char buffRead[BUFSIZE + 1];
      int total = 0;

      int randNumber = rand() % 10; //for login and register
functionalities
```

```
	/**
	 * 0: get_exhibitions
	 * 1: get_pieces
	 * 2: register
	 * 3: login
	*/

	switch(action) {
		case 0:
			printf("Testing get_exhibitions\n");
			sprintf(buffWrite, "get_exhibitions");

			send(sockfd, buffWrite, sizeof(buffWrite), 0);
			memset(buffRead, 0, BUFSIZE + 1);
			memset(buffWrite, 0, BUFSIZE + 1);

			do{
				memset(buffRead, 0, BUFSIZE + 1);
				total = recv(sockfd, buffRead, sizeof(buffRead),
0);
				printf("From Server : %s\n", buffRead);
			}while(total > 0);

			/*Resetting*/
			memset(buffRead, 0, BUFSIZE + 1);
			memset(buffWrite, 0, BUFSIZE + 1);
			break;

		case 1:
			printf("Testing get_pieces\n");
			sprintf(buffWrite, "get_pieces 1 SINGLE:STANDARD");

			send(sockfd, buffWrite, sizeof(buffWrite), 0);
			memset(buffRead, 0, BUFSIZE + 1);
			memset(buffWrite, 0, BUFSIZE + 1);

			do{
				memset(buffRead, 0, BUFSIZE + 1);
				recv(sockfd, buffRead, sizeof(buffRead), 0);
				printf("From Server : %s\n", buffRead);
			}while(total > 0);

			/*Resetting*/
			memset(buffRead, 0, BUFSIZE + 1);
			memset(buffWrite, 0, BUFSIZE + 1);

			break;

		case 2:
			printf("Testing register\n");

			sprintf(buffWrite, "register %s:%s:%s",
```

```
users[randNumber].name, users[randNumber].email,
users[randNumber].password);
                printf("BUFFER %s\n", buffWrite);
                send(sockfd, buffWrite, sizeof(buffWrite), 0);

                memset(buffRead, 0, BUFSIZE + 1);
                memset(buffWrite, 0, BUFSIZE + 1);

                recv(sockfd, buffRead, sizeof(buffRead), 0);
                printf("From Server : %s\n", buffRead);

                /*Resetting*/
                memset(buffRead, 0, BUFSIZE + 1);
                memset(buffWrite, 0, BUFSIZE + 1);
                break;

           case 3:
                printf("Testing login\n");

                sprintf(buffWrite, "login %s:%s",
users[randNumber].email, users[randNumber].password);

                send(sockfd, buffWrite, sizeof(buffWrite), 0);

                memset(buffRead, 0, BUFSIZE + 1);
                memset(buffWrite, 0, BUFSIZE + 1);

                recv(sockfd, buffRead, sizeof(buffRead), 0);
                printf("From Server : %s\n", buffRead);

                /*Resetting*/
                memset(buffRead, 0, BUFSIZE + 1);
                memset(buffWrite, 0, BUFSIZE + 1);
                break;
     }
}

void client(int action) {
     int sockfd;
     struct sockaddr_in servaddr;

     // socket creation and verification
     sockfd = socket(AF_INET, SOCK_STREAM, 0);

     if (sockfd == -1) {
          perror("Socket creation failed...\n");
          exit(EXIT_FAILURE);
     } else
          printf("Socket successfully created..\n");

     bzero(&servaddr, sizeof(servaddr));
```

```
	// assign IP, PORT
	servaddr.sin_family = AF_INET;
	servaddr.sin_addr.s_addr = inet_addr("127.0.0.1");
	servaddr.sin_port = htons(PORT);

	// connect the client socket to server socket
	if (connect(sockfd, (struct sockaddr*)&servaddr, sizeof(servaddr))
!= 0) {
		perror("Connection with the server failed...\n");
		exit(EXIT_FAILURE);
	}else
		printf("Connected to the server.. socket %d\n", sockfd);

	func(sockfd, action);

	// close the socket
	close(sockfd);
}

int main(int argc, char *argv[]) {
	int count;
	int lower = 0, upper = 3;

	if(argc != 2) {
		printf("Usage: ./client <number of clients>");
	exit(EXIT_FAILURE);
	}

	srand(time(0));
	count = strtol(argv[1], NULL, 10);

	for(int i = 0; i<count; i++) {
		if(fork() == 0) {
			client(i % 4);
			exit(0);
		}
	}

	for(int i = 0; i<count; i++)
		wait(NULL);

	return 0;
}
```