



Università degli Studi di Napoli
FEDERICO II

A.A. 2022-2023

INFOPOINT



Lucia Brando



l.brand@studenti.unina.it



N86003382



Dario Morace



da.morace@studenti.unina.it



N86003778



Valentino Bocchetti



vale.bocchetti@studenti.unina.it



N86003405

INDICE

0 SU INFOPOINT	3
0.1 Presentazione	4
1 GUIDA ALL'USO	5
1.1 Guida al Server	6
1.1.1 Funzionalità	6
1.1.2 Scelte implementative	6
1.1.3 Tecnologie e strumenti utilizzati	6
1.1.4 Memorizzazione dei dati	6
1.2 Guida al Client	7
1.2.1 Primo avvio	7
1.2.2 Post registrazione	7
1.2.3 Memorizzazione delle informazioni	7
1.2.4 Modelli di dominio	8
2 PROTOCOLLO APPLICATIVO	9
3 DETTAGLI IMPLEMENTATIVI	10
3.1 Server	11
3.1.1 Struttura del server	11
3.1.2 File di configurazione	12
3.1.3 Compilazione	12
3.1.4 Esecuzione	13
3.1.5 Analisi del codice sviluppato	13
3.2 Client	17
3.2.1 Struttura del client	17
3.2.2 Compilazione ed Esecuzione	17
3.2.3 Analisi del codice sviluppato	18
4 CODICE SORGENTE SVILUPPATO	23
5 RINGRAZIAMENTI	24

CAPITOLO: SU INFOPOINT

ESTRATTO

InfoPoint[®] è una piattaforma che nasce per offrire un supporto ai visitatori del museo.

L'implementazione è suddivisa in 2 componenti che identificheremo come **server** e **client**

- Un backend scritto in C per la gestione dei dati, che fa da server, con la possibilità di essere hostato¹ sia su bare metal sia in maniera containerizzata²;
- Una applicazione Android, scritta in Java che fa da client;

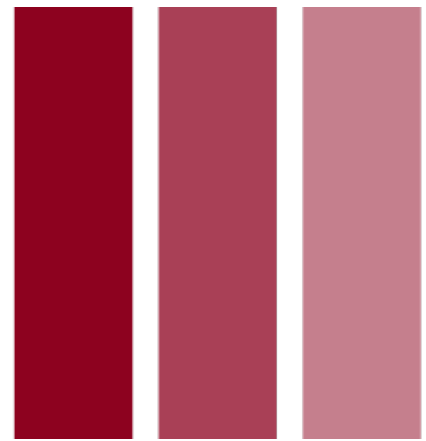
¹Indica un servizio di rete che consiste nell'allocare su un server web delle pagine web di un sito web o di un'applicazione web, rendendolo così accessibile dalla rete Internet e ai suoi utenti

²È messa a disposizione una immagine docker plug & play



INFOPOINT

Educate Yourself



1

CAPITOLO: GUIDA ALL'USO

ESTRATTO

Come detto **InfoPoint** è suddiviso nelle 2 componenti di **server** e **client**. Di seguito riportiamo, per entrambi, l'analisi e le scelte effettuate durante il loro sviluppo



1.1.1 ■ Funzionalità

Il Sistema, deve offrire, una serie di funzionalità:

- Possibilità di connessione concorrente;
- Possibilità di potersi registrare alla piattaforma¹;
- Possibilità di usufruire dei contenuti in base alla tipologia di utente, in modo da permettere un focus diverso in base alle sue caratteristiche²;

1.1.2 ■ Scelte implementative

Seguendo il concetto del *DIVIDE ET IMPERA*³ si è scelto di spezzare le varie funzionalità che vengono messe a disposizione per rendere il codice facilmente manutenibile ed evitare lo stato di codice monolitico⁴.

In particolare l'implementazione fa ampiamente uso di codice sviluppato in **C POSIX**, che permette l'utilizzo di funzionalità e **system call**⁵ su cui si basa l'intera code-base: send, recv, socket, thread pool, mutex, argp, ...

1.1.3 ■ Tecnologie e strumenti utilizzati

Per una migliore gestione del Sistema, si è fatto uso di una serie di strumenti.

Durante lo sviluppo si è fatto uso dell'utility **cmake**⁶, tool modulare che permette la generazione di un **Makefile**⁷ automatizzato.

Per la fase di deploy invece si è fatto uso di **docker**, tool che permette l'esecuzione di programmi in maniera containerizzata.

Come sperato, avendo adottato entrambe le strategie non si sono riscontrati problemi durante il passaggio da un ambiente locale (di testing) a uno decentralizzato (production ready).

1.1.4 ■ Memorizzazione dei dati

Per essere sempre in linea con le nuove tendenze e tecnologie si è scelto di abbandonare il classico approccio basato su un collegamento ad una base di dati relazionale, preferendo un approccio di tipo **NOSQL**⁸, che offre una maggiore elasticità⁹ e scalabilità¹⁰ nel tempo.

¹Le credenziali vengono salvate facendo uso di un Database, che risulta molto più affidabile di un semplice file di testo

²Ricordiamo che il bacino degli utenti che possono fare uso del sistema può variare da scolaresche, famiglie o esperti

³Metodologia per la risoluzione di problemi → Il problema viene diviso in sottoproblemi più semplici e si continua fino a ottenere problemi facilmente risolvibili. Combinando le soluzioni ottenute si risolve il problema originario.

⁴Che risulta notoriamente più difficile da gestire e modificare nel tempo

⁵Una chiamata si definisce, appunto, **di sistema**, quando fa uso di servizi e funzionalità a livello kernel del sistema operativo in uso.

⁶Per maggiori informazioni visitare il seguente [link](#)

⁷Che contiene tutte le direttive utilizzate dall'utility make, che ne permettono la corretta compilazione

⁸Che a differenza dei classici DBMS relazionali (che offrono un approccio **relazione** ai dati) offre un approccio al documento, rendendo il design più semplice

⁹Per loro natura infatti sono pensati per lavorare in situazioni di alto carico di lavoro pur mantenendo basse latenze

¹⁰Per la loro natura offrono una forte resilienza a situazioni di fault, considerando il focus scelto (AP) nel **CAP** (Consistency-Availability-Partition Tolerance)



Prestando particolare attenzione alla semplicità di utilizzo che un'applicazione mobile deve garantire si è scelto di fare uso di un'interfaccia snella e lineare.

1.2.1 Primo avvio

All'avvio all'utente è richiesto di registrarsi o accedere alla piattaforma, facendo uso della classica combinazione mediante **username** e **password**.

1.2.2 Post registrazione

In seguito ad una corretta registrazione/accesso all'utente viene mostrata la **HomePage**, nella quale ha la possibilità di visualizzare:

- ▶ L'elenco delle **artwork** presenti nel museo;
- ▶ La possibilità di poter ricercare le opere in base a
 - ◊ Nome;
 - ◊ Autore;
 - ◊ Data;
- ▶ Il proprio profilo e le relative informazioni;
- ▶ L'accesso diretto alle artwork preferite.

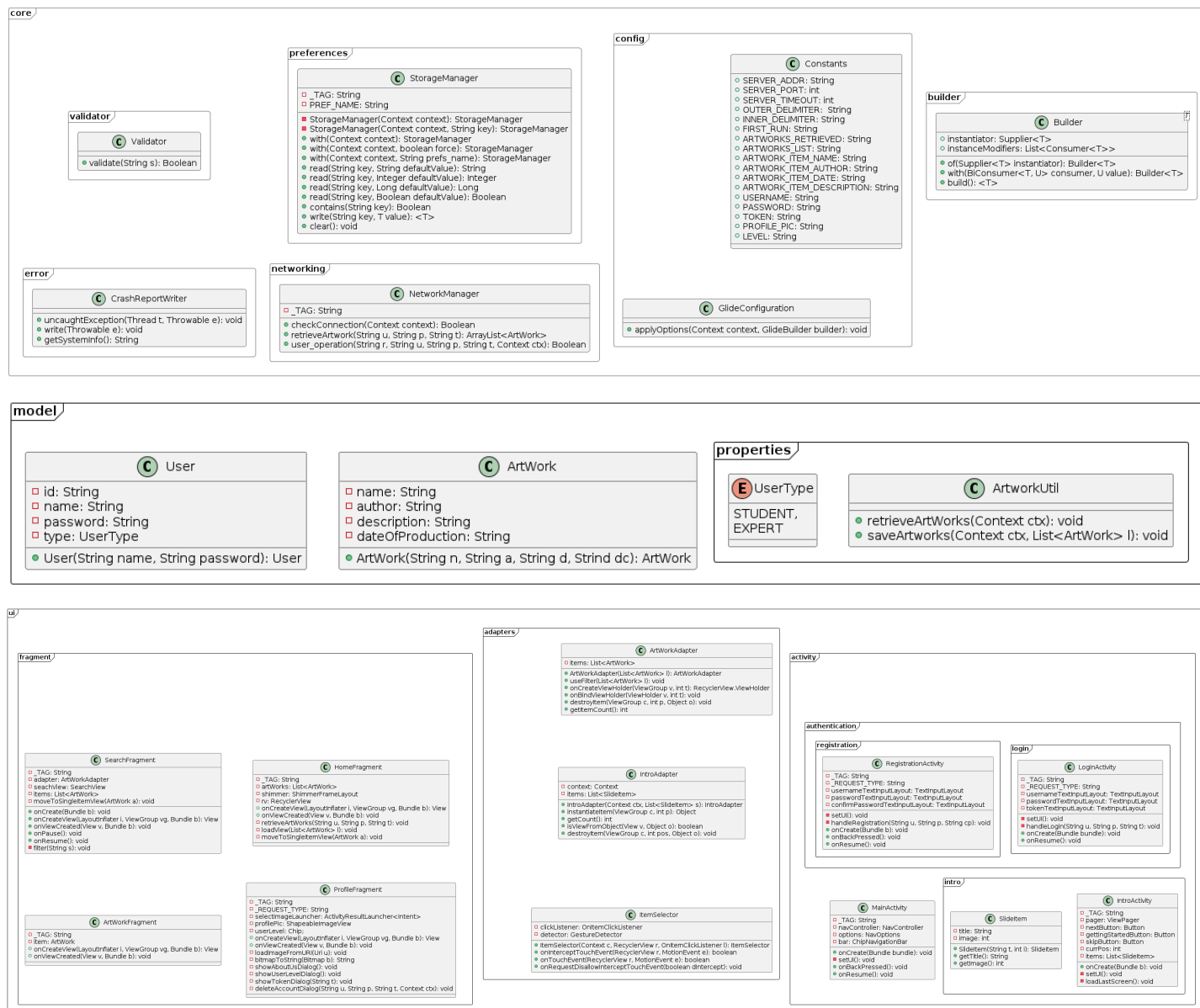
1.2.3 Memorizzazione delle informazioni

In linea con il design pattern **MVVM**¹¹, si è fatto uso delle **SharedPreferences** per il salvataggio e la gestione di componenti chiave quali:

- ▶ Credenziali dell'utente;
- ▶ Variabili d'ambiente;
- ▶ Variabili di stato;

¹¹(Model-view-viewmodel), pattern nel quale viene astratto lo stato di view (visualizzazione) e comportamento

Di seguito riportiamo il diagramma delle classi di Analisi prodotto durante lo sviluppo della piattaforma InfoPoint



2

CAPITOLO: PROTOCOLLO APPLICATIVO

Come già indicato in precedenza abbiamo preferito il protocollo **TCP** rispetto al protocollo **UDP**, per la presenza di un controllo della congestione e affidabilità in termini di invio/ricezione di dati¹.

Lo sviluppo dell'applicativo è stato inizialmente verticalizzato sulla creazione dello scheletro del Server, per avere un primo approccio nudo e crudo allo scambio di messaggi via **socket**.

Per avere un programma robusto e manutenibile si è fatto largo uso delle **good practices** che questo tipo di comunicazione richiede. In particolare:

- ▶ La connessione viene aperta solo nel momento in cui devono essere inviati/ricevuti dati (Si evita in questo modo di tenere aperte connessioni in momenti in cui queste non vengono sfruttate);
- ▶ Si effettuano controlli di raggiungibilità del server lato client²;
- ▶ Vengono effettuati controlli e gestione degli stati di tutte le operazioni lato **Server**;
- ▶ Tutti i dati scambiati tra **Server** e **Client** vengono opportunamente controllati³, onde evitare operazioni che possano minare il corretto funzionamento dell'applicativo

¹Ricordiamo infatti che UDP non ha garanzie sulla trasmissione dei pacchetti, seguendo la logica di best-effort

²Non ha senso infatti tenere aperta una connessione se non utilizzata, anzi si rischia anche di causare interruzione di servizio dovuti a timeout improvvisi

³Si fa uso infatti di **TAG** per la parametrizzazione dei messaggi inviati/ricevuti per una maggiore sicurezza

3

CAPITOLO: DETTAGLI IMPLEMENTATIVI

ESTRATTO

In questo capitolo tratteremo l'implementazione e il funzionamento delle componenti che hanno reso possibile lo sviluppo della piattaforma **InfoPoint**, prestando particolare attenzione alle funzionalità richieste da programma

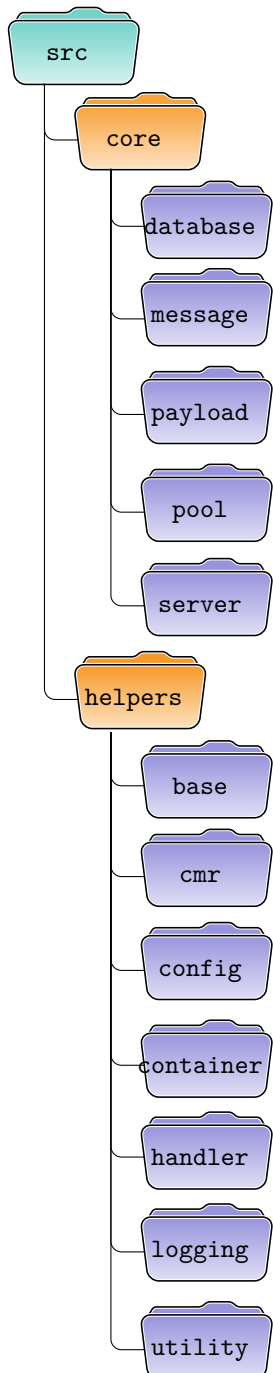


Di seguito riportiamo alcuni dettagli implementativi della struttura del **Server**. In particolare:

- Analisi della struttura del progetto;
- Analisi e controllo delle principali system call che il C mette a disposizione per la programmazione tramite socket¹;
- Analisi delle funzioni di logging e controllo degli stati utilizzate;
- Analisi e controllo delle principali system call per la gestione del Database utilizzato;
- Analisi e gestione della coda dei client connessi che richiedono uno scambio di informazioni.

3.1.1 ■ Struttura del server

La struttura del server è così strutturata:



Core

(Contiene le componenti fondamentali alla definizione del server)

- **server** → È il cuore dell'applicativo. Svolge il compito di orchestratore di tutto il programma;
- **database** → Contiene la struttura e la logica del gestore (handler) che ha il compito di comunicare con l'istanza del Database NOSQL MongoDB;
- **message** → Contiene delle funzioni wrapper utili alla comunicazione tramite socket (di cui l'applicativo fa largo uso);
- **payload** → Prendendo ispirazione dal protocollo HTTP facciamo uso del concetto di Payload, oggetto che rappresenta informazioni (su Utenti e Opere) che i client e il server si scambieranno.

Helpers

(Contiene le componenti necessarie alla definizione dell'infrastruttura)

- **base** → Contiene la definizione di
 - ◊ Alias di tipi comuni (sfruttando la keyword typedef) per una più facile modifica nel tempo;
 - ◊ Macro per operazioni elementari;
- **cmr** (command_line_runner) → Fa da wrapper alla libreria argp.h per il parsing dei parametri passati all'eseguibile prodotto in fase di compilazione;
- **config** → Ha il compito di recuperare le informazioni presenti all'interno del file di configurazione (in formato ini) messo a disposizione per il settaggio delle opzioni necessarie all'esecuzione del server;
- **pool** → Thread-Pool utilizzata dal server per soddisfare le richieste in ingresso;
- **utility** → Contiene la definizione di funzioni per
 - ◊ Lettura e recupero informazioni di file;
 - ◊ Definizione di Regex utilizzate dal programma;
 - ◊ Manipolazione di buffer (trimming e concatenazione).

¹Per un'analisi più completa si faccia riferimento al codice sorgente, come indicato nel capitolo successivo

3.1.2 ■ File di configurazione

Il server mette a disposizione la possibilità di essere configurato mediante l'uso di un file di configurazione in formato **ini** che permette di customizzare ogni aspetto del server. Di seguito riportiamo il suo contenuto:

```
1  ;;;; Info Point configuration file ;;;;
2
3  ;; [Network] How the server should setup network
4  [network]
5
6  ; Bind to a single interface
7  ; If not specified all the interfaces will listen for incoming connections.
8  host = localhost
9
10 ; Accept connections on the specified port, default is 9090.
11 ; If port 0 is specified the server will not listen on a TCP socket.
12 port = 9090
13
14 ; Close the connection after a client is idle for N seconds (0 to disable)
15 timeout = 0
16
17 ;; [Connections] How the server should handle connections using the specified thread pool ;;
18 [connections]
19
20 ; Set the max number of connected clients at the same time.
21 ; By default this limit is set to 100 clients.
22 ; Once the limit is reached the server will close all the new connections.
23 max_clients = 5
24
25 ; Set the max number of worker threads. (Set this to 0 to automate it)
26 max_threads = 5
27
28 ;; [Logging] How & where the server should log
29 [logging]
30 ; Set server verbosity. Available options:
31 ; + debug (a lot of information, useful for development/testing)
32 ; + verbose (many rarely useful info, but not a mess like the debug level)
33 ; + notice (moderately verbose, what you want in production probably)
34 ; + warning (only very important / critical messages are logged)
35 log_level = debug
36
37 ; Specify the log file name.
38 ; Also 'stdout' can be used to force the server to log on the standard output.
39 ; Note that if you use standard output for logging but daemonize,
40 ; logs will be sent to /dev/null
41 log_file = stdout
42
43 ;; [Database] How the server should connect & communicate with the database
44 [database]
45 ; Type of the database to connect with.
46 ; Currently only mongodb is supported
47 type = mongodb
48 ; Username used to authenticate
49 username = admin
50 ; Password used to authenticate
51 password = password
52 ; Host on which the database is running
53 host = localhost
54 ; Port on which the database is listening to
55 port = 27017
56 ; Name of the database used
57 name = infopoint
```

▢ [../server/src/info_point.ini](#)

3.1.3 ■ Compilazione

Tutti i file sorgente necessari alla compilazione del server sono contenuti nella directory **server/src**.

Il processo risulterà particolarmente semplificato in quanto, come ampiamente presentato nelle sezioni precedenti, faremo totale affidamento allo strumento **cmake**.

Con un **terminal emulator** e una shell qualsiasi andiamo quindi ad eseguire i seguenti comandi:

```
# Per costruire le informazioni necessarie a cmake per la futura compilazione
```

```
cmake -S . -B build/ -DCMAKE_EXPORT_COMPILE_COMMANDS=ON
# Per compilare il server
cmake --build build/
# Per l'operazione di clean up dei file generati da CMake
cmake --build build/ --target clean
```



L'output ottenuto sarà da ricercare nella directory `bin/` sotto il nome di **InfoPointServer**².

3.1.4 Esecuzione

L'utilizzo del server prevede la sola esecuzione dell'eseguibile in quanto è un sistema totalmente autonomo.

Attraverso l'uso della libreria `argp.h`³ il server mette a disposizione una serie di flags in modo da poter:

- Personalizzare l'esecuzione del server stesso;
- Avere informazioni
 - ◊ Sulla versione dell'applicativo;
 - ◊ Attraverso **Help Menu** stile man;

```
# Esecuzione del server utilizzando il file di configurazione
./build/InfoPointServer -c src/info_point.ini
# Esecuzione del server utilizzando facendo uso della configurazione di default (-d o -use-default-config)
./build/InfoPointServer -d
# Esecuzione del server per mostrare l'help menu
./build/InfoPointServer --help/--usage/-?
# Esecuzione del server per ottenere informazioni sulla versione del server
./build/InfoPointServer -V
```



3.1.5 Analisi del codice sviluppato

In questa fase andremo quindi ad analizzare nello specifico le strutture e il codice sviluppato per l'esecuzione del server, a partire dal chiamante (📄 `main`):

```
1 #include "core/server/server.h"
2 #include "helpers/config/info_point_config.h"
3 #include "helpers/command_line_runner/command_line_runner.h"
4
5 int main(int argc, char** argv) {
6     // Welcome message
7     fprintf(stdout, ANSI_COLOR_BMAGENTA "%s" ANSI_COLOR_RESET "\n", welcome_msg);
8
9     // Parse command line arguments
10    char* config_file = parse_command_line_arguments(argc, argv);
11
12    // Initialize the configuration structure based on how the user invoke the server:
13    // + If no config file is passed (server invoked with the -d flag) → Populate the configuration using a
14    //   default one
15    // + If a config file is passed (server invoked with the -c <FILE>) → Populate the configuration using
16    //   the given <FILE>
17    info_point_config* cfg = (config_file != NULL) ? provide_config(config_file) : provide_default_config();
18
19    // Show the settings getted from the file
20    cfg_pretty_print(cfg, stdout);
21
22    server* s = init_server(cfg->ns.port, cfg->cs.max_clients, cfg->cs.max_workers, cfg->ds.username, cfg->
23    ds.password, cfg->ds.host, cfg->ds.database_name);
24
25    // Finally free the cfg, as we no more need it
26    free(cfg);
27
28    server_loop(s);
29
30    destroy_server(s);
31 }
```



Payload

²Modificabile dal file di configurazione di cmake

Per una più semplice gestione nel passaggio delle informazioni **server** \iff **client** si è pensato di utilizzare una classe wrapper che andasse a simulare il concetto di **Payload**, costituita da:

- Tipo di richiesta;
- Contenuto effettivo del payload³
- Dimensione del payload.

```
1 typedef enum request_t {
2     char* request_type;           // Type of the request: One of LOGIN, REGISTRATION, RETRIEVE, DELETE
3     char* token;                  // Token of the caller
4     char* credential_username;    // Username of the caller
5     char* credential_password;    // Password of the caller
6 } request_t;
7
8 typedef struct payload_t {
9     void* data; // Content of the payload
10    size_t size; // Actual size
11 } payload_t;
```



Per una semplice manipolazione del dato viene offerta la funzione **parse_data**⁴ che effettua il parsing in base a 2 delimitatori (personalizzabili) e una funzione wrapper **destroy_request**⁵ per l'eliminazione delle risorse di una richiesta appena soddisfatta.

Database Handler

La struct **db_handler** è da considerarsi una "classe" wrapper⁴ → Fa da tramite per la comunicazione con il database NOSQL MongoDB.

Di seguito ne riportiamo una descrizione ad alto livello:

```
1 typedef struct db_handler {
2     struct instance {
3         mongoc_client_pool_t* pool; // Connection pool for multi-threaded programs
4         mongoc_uri_t* uri;          // Abstraction on top of the MongoDB connection URI format
5     } instance;
6
7     // Structure representing information used to:
8     // Connect & authenticate to a given mongodb instance
9     // Do basic operation with the specified documents used by the service
10    struct mongo_db_settings {
11        char name[100]; // Database name identifier
12        char database_uri[100]; // String used to connect to the mongodb instance
13        char* user_collection; // Identifier of the collection used to store & retrieve data of the users
14        char* art_work_collection; // Identifier of the collection used to store & retrieve data of the
15                                   artworks
16    } settings;
17 } db_handler;
```



A supporto di questa struttura sono presenti 2 strutture che rappresentano le collezioni utilizzate dal server⁵:

```
1 typedef struct art_work {
2     char* name;
3     char* author;
4     char* date;
5     char* description;
6 } art_work;
7
8 typedef struct user {
9     char* id;
10    char* name;
11    char* password;
12    char* level;
13 } user;
```



³Per una generalizzazione massima del contenuto si fa uso del tipo **void***

⁴Per wrapper si intende un oggetto che incapsula funzionalità di più basso livello affinché possa semplificarne l'utilizzo ai fini produttivi

⁵Per entrambe sono presenti funzioni per l'inizializzazione e eliminazione delle risorse da queste utilizzate

Seguendo le **good practices** per la programmazione di una corretta API vengono espone una serie di metodi che vengono di seguito descritte:

- ▶ **init_handler**⁶ → Ha il compito di inizializzare delle risorse dell'handler ad esso associato;
- ▶ **destroy_handler**⁶ → Ha il compito di liberare le risorse dell'handler ad esso associato;
- ▶ **populate_collection**⁶ → Ha il compito di popolare la collezione delle opere con i dati prestabili;
- ▶ **retrieve_art_works**⁶ → Ha il compito di recuperare dal database associato tutte le informazioni presenti relative alle opere (artworks);
- ▶ **is_present**⁶ → Ha il compito di controllare se il documento specificato sia presente o meno nella collezione specificata;
- ▶ **insert_single**⁶ → Ha il compito di inserire il documento specificato nella collezione specificata controllando lo stato dell'operazione;
- ▶ **delete_single**⁶ → Speculare ad **insert_single**⁶ ha il compito di eliminare il documento specificato nella collezione specificata controllando lo stato dell'operazione;
- ▶ **parse_bson_as_artwork**⁶ → Ha il compito di convertire il documento **bson_t** (binary json format) ⇔ ad **artwork** wrappata nella struttura **payload_t**
- ▶ **parse_bson_as_user**⁶ → Ha il compito di convertire il documento **bson_t** (binary json format) ⇔ ad **user** wrappata nella struttura **payload_t**
- ▶ **test_connection**⁶ → Ha il compito di testare se la connessione al database definito sia valida o meno.

Thread Pool

Per una gestione concorrente e parallela il server fa uso di una **Thread Pool** che attende che uno specifico lavoro venga inserito nella coda di task da eseguire. La struttura è così definita:

```
1 typedef struct thread_pool_t {
2     pthread_t* threads; // Array of threads spawned by the thread pool
3     size_t threads_alive; // N° of threads currently active
4
5     pthread_mutex_t lock; // Mutex of the thread pool
6     pthread_cond_t signal; // Conditional variable for the thread pool
7     queue_t* queue; // Queue containing all the tasks that has to be executed
8
9     bool active; // Control switch for the worker threads
10    bool on_shutdown; // Control switch for the worker threads in order to signal shutdown
11    db_handler* handler; // Database communication handler
12
13 } thread_pool;
```



Analogamente alle altre strutture anche per la Thread Pool vengono offerti dei metodi di supporto:

- ▶ **init_thread_pool**⁶ → Ha il compito di inizializzare delle risorse della thread pool ad esso associato;
- ▶ **destroy_thread_pool**⁶ → Ha il compito di liberare le risorse della thread pool ad esso associato;
- ▶ **submit_task**⁶ → Ha il compito di aggiungere alla coda di lavori della thread pool un nuovo task da eseguire;
- ▶ **execute_task**⁶ → È la funzione eseguita da tutti i threads spawnati dalla Thread Pool;
- ▶ **handle_request**⁶ → Ha il compito di gestire le richieste ricevute dai vari client ⁶;

Message

Mette a disposizione 2 funzioni utili per un rapido invio/ricezione di dati mediante socket.

Server

È l'orchestratore di tutto l'applicativo. Fa da wrapper per tutte le strutture sopra citate, rendendo più snella e semplice la gestione dell'intera struttura del server. È così definito:

```
1 typedef struct server {
2     ssize_t socket;
3     struct sockaddr_in transport;
4     thread_pool* pool;
5 } server;
```



Anche in questo caso sono presenti dei metodi di supporto all'utilizzo della struttura:

- ▶ **init_server**⁶ → Ha il compito di inizializzare delle risorse del server ad esso associato;
- ▶ **destroy_server**⁶ → Ha il compito di liberare le risorse del server ad esso associato;

⁶Validazione dei dati inviati, contatto con il database in esecuzione, ...

- ▶ **server_loop**⁸ → Ha il compito di inizializzare l'handler per la terminazione "gracefully" del programma e fare da dispatcher per l'arrivo di nuove richieste di connessione;

Finito l'analisi della struttura centrale del server passiamo ora all'analisi delle strutture di supporto di cui le prime fanno uso.

Base

Contiene la definizione di:

- ▶ Macro;
- ▶ Abbreviazioni di tipi.

Command Line Runner

Classe wrapper per la libreria **argp.h**⁹

Config

Classe utility che:

- I. Effettua il parsing del file di configurazione (se specificato);
- II. Costruisce la configurazione in base ai dati ottenuti.

Container

Contiene la definizione e la struttura della **queue**⁷ utilizzato dalla thread pool per la gestione dei task. È così definita

```
1 typedef struct node {
2     struct node* next; // Reference to the next node
3     void* data; // Content of the node
4 } node_t;
5
6 typedef struct queue {
7     node_t* head; // Reference to the head of the queue
8     node_t* tail; // Reference to the tail of the queue
9     size_t size; // Size of the queue
10 } queue_t;
```



Per un più semplice utilizzo la struttura offre i seguenti metodi di supporto:

- ▶ **init_queue**⁸ → Ha il compito di inizializzare delle risorse della queue ad essa associata;
- ▶ **destroy_queue**⁸ → Ha il compito di liberare le risorse della queue ad essa associata;
- ▶ **enqueue**⁸ → Ha il compito di inserire un nodo all'interno della queue;
- ▶ **enqueue**⁸ → Ha il compito di recuperare un nodo all'interno della queue;
- ▶ **retrieve_queue_size**⁸ → Ha il compito di recuperare la dimensione corrente della queue;
- ▶ **is_empty**⁸ → Ha il compito di controllare se la queue sia vuota o meno;

Handler

Contiene la definizione per l'inizializzazione dell'handler utilizzato dal server per la terminazione in caso di ricezione di un determinato segnale (in questo caso si è utilizzato il segnale **SIGINT**).

Logging

Contiene la definizione delle funzioni necessarie per il logging delle informazioni, che permea in tutto il progetto.

Si noti come si fa largo uso di funzioni variadiche, permettendo una struttura dinamica del logging.

È presente inoltre anche un messaggio di benvenuto generato mediante l'utility **xxd**

Utility

Contiene funzioni di supporto generiche utilizzate all'interno del progetto, tra cui:

- ▶ Funzioni di **trimming**;
- ▶ Validazione di **regex**;
- ▶ Operazioni su file;
- ▶ Operazioni su "stringhe";

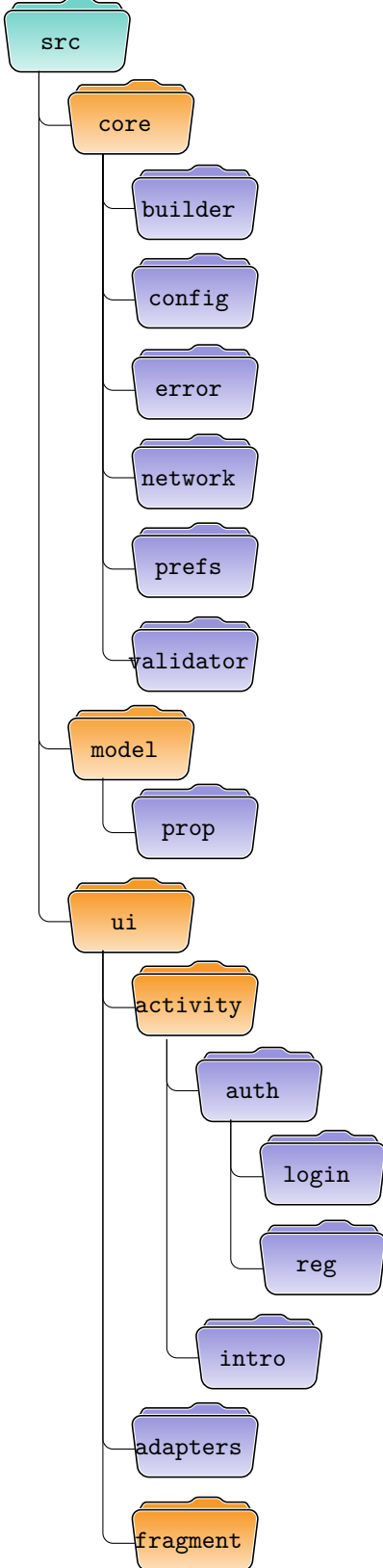
⁷Per ovvi motivi risulta comodo utilizzare una struttura di tipo FIFO (First In - First Out)



Di seguito riportiamo alcuni dettagli implementativi della struttura del **Client**.

3.2.1 ■ Struttura del client

La struttura del client è così strutturata:



Core

(Contiene le componenti fondamentali alla definizione del client)

- ▶ **builder** → Offre supporto alla costruzione di oggetti mediante Generics (Generic Builder);
- ▶ **config** → Contiene tutte le informazioni necessarie alla configurazione dell'applicativo;
- ▶ **error** → Offre supporto alla gestione degli errori che possono avvenire durante l'esecuzione dell'applicazione (Mette a disposizione una api per la gestione dello stack trace);
- ▶ **network** (networking) → Offre supporto alle operazioni che richiedono l'utilizzo della rete
- ▶ **prefs** (preferences) → Offre il supporto alle shared preferences in maniera semplice e sicura
- ▶ **validator** → Offre il supporto alla validazione di Username e Password inserite dall'utente

Model

(Contiene le componenti necessarie alla definizione dei modelli utilizzati e loro proprietà)

UI

(Contiene le componenti necessarie alla definizione dell'infrastruttura grafica)

- ▶ **activity** → Contiene la definizione delle activity⁸ per
 - ◊ Autenticazione, ovvero
 - Login;
 - Registrazione (reg)
 - ◊ Esposizione dell'applicazione (Intro)
- ▶ **adapters** → Offre gli adapters⁹ utilizzati dalle activity per la gestione delle componenti grafiche che fanno uso di componenti oggetto
- ▶ **fragments** → Contiene la definizione di tutti i fragment¹⁰ presenti nell'applicazione

3.2.2 ■ Compilazione ed Esecuzione

Entrambe le operazioni sono effettuabili dall'IDE **Android Studio** in seguito all'apertura del progetto¹¹.

¹⁰La "finestra" grafica con cui l'utente interagisce

¹⁰Un ponte per il componente grafico e gli oggetti che lo compongono


¹⁰La porzione "riusabile" della UI dell'applicazione

¹¹Per maggiori informazioni visitare il seguente [link](#)

3.2.3 ■ Analisi del codice sviluppato

In questa fase andremo quindi ad analizzare nello specifico i concetti chiave alla base dell'applicazione Android che fa da client.

SharedPreferences

Le **SharedPreferences** sono fondamentali per la memorizzazione delle informazioni all'interno di una applicazione Android. Proprio per questo motivo si è scelto di farne uso facendo affidamento alla classe wrapper  **StorageManager** da noi così strutturata:

```
1 public class StorageManager {
2     private static final String _TAG = "[StorageManager] ";
3     private static final String PREF_NAME = "app_settings";
4     private static StorageManager instance;
5     private static SharedPreferences preferences;
6
7     private StorageManager(@NonNull Context context) {
8         preferences = context.getApplicationContext().getSharedPreferences(PREF_NAME, Context.MODE_PRIVATE);
9     }
10
11     private StorageManager(@NonNull Context context, String key) {
12         preferences = context.getApplicationContext().getSharedPreferences(key, Context.MODE_PRIVATE);
13     }
14
15     public static StorageManager with(@NonNull Context context) {
16         if (instance == null)
17             instance = new StorageManager(context);
18         return instance;
19     }
20
21     public static StorageManager with(@NonNull Context context, boolean forceInstantiation) {
22         if (forceInstantiation)
23             instance = new StorageManager(context);
24         return instance;
25     }
26
27     public static StorageManager with(@NonNull Context context, String preferencesName) {
28         if (instance == null)
29             instance = new StorageManager(context, preferencesName);
30         return instance;
31     }
32
33     public String read(String key, String defaultValue){return preferences.getString(key, defaultValue);}
34
35     public Integer read(String key, Integer defaultValue){return preferences.getInt(key, defaultValue);}
36     public Long read(String key, Long defaultValue){return preferences.getLong(key, defaultValue);}
37     public Boolean read(String key, Boolean defaultValue){return preferences.getBoolean(key, defaultValue);}
38     ;}
39
40     public boolean contains(String key){ return preferences.contains(key); }
41
42     public <T> void write(String key, T value) {
43         SharedPreferences.Editor editor = preferences.edit();
44
45         // Retrieve the type of value using the simple name of the underlying class as given in the source
46         // code
47         // avoiding using instanceof for each possible type (Supported type are String, Integer, Long,
48         // Boolean)
49         switch (value.getClass().getSimpleName()) {
50             case "String" → editor.putString(key, (String) value);
51             case "Integer" → editor.putInt(key, (Integer) value);
52             case "Long" → editor.putLong(key, (Long) value);
53             case "Boolean" → editor.putBoolean(key, (Boolean) value);
54             default → Log.d(_TAG, "Unknown type <" + value.getClass().getSimpleName() + "> was given");
55         }
56         editor.apply();
57     }
58
59     public void clear() { preferences.edit().clear().apply(); }
```



Modelli

Rappresentazione ad Oggetti del concetto di Utente ed Opere e le loro rispettive proprietà.

Vengono utilizzati come contenitori di conversione dei dati passati tra client e server. Sono così strutturati:

```
1 public class User {
2     private String id;
3     private String name;
4     private String password;
5 }
```

▢ ../../client/app/src/main/java/com/infopoint/model/User.java




```
1 public class ArtWork {
2     private String name;
3     private String author;
4
5
6     private String dateOfProduction;
7     private String description;
8 }
```

▢ ../../client/app/src/main/java/com/infopoint/model/ArtWork.java

Properties

Per una migliore gestione dei Models si è scelto di utilizzare delle classi di supporto.

In particolare:

- ▶  **ArtworkUtil** → Ha il compito di gestire tutte le opere sfruttando l'uso delle Shared Preferences mediante la classe  **StorageManager**
- ▶  **UserType** → Permette di descrivere il livello dell'utente loggato

Di seguito riportiamo la loro struttura:

```
1 public class ArtworkUtil {
2     private final static String _TAG = "[ArtworkUtil] ";
3     public static ArrayList<ArtWork> retrieveArtWorks(Context ctx) {
4         Log.d(_TAG, "Retrieving artworks ... ");
5         String key = StorageManager.with(ctx).read(Constants.ARTWORKS_LIST, "");
6         Gson gson = new GsonBuilder().create();
7         return gson.fromJson(key, new TypeToken<ArrayList<ArtWork>>() {}.getType());
8     }
9     public static void saveArtworks(Context context, List<ArtWork> artworks) {
10        Log.d(_TAG, "Saving artworks ... " + artworks.get(1).getDateOfProduction());
11        Gson gson = new GsonBuilder().create();
12        String value = gson.toJson(artworks);
13        StorageManager.with(context).write(Constants.ARTWORKS_LIST, value);
14        StorageManager.with(context).write(Constants.ARTWORKS_RETRIEVED, true);
15    }
16 }
```

▢ ../../client/app/src/main/java/com/infopoint/model/properties/ArtworkUtil.java

```
24 public enum UserType { STUDENT, EXPERT }
```

▢ ../../client/app/src/main/java/com/infopoint/model/properties/UserType.java

Ui

Come qualsiasi applicazione che si rispetti, si è voluto porre un focus importante sulla produzione di una interfaccia semplice e intuitiva.

A questo si aggiunge una serie di funzioni di supporto per:

- ▶ Autenticazione;
- ▶ Recupero delle informazioni e loro manipolazione;
- ▶ Operazioni di supporto all'utente.
- ▶ Controllo dell'accesso alla rete (fondamentale per il corretto funzionamento);

Di queste operazioni riportiamo alcuni estratti che riteniamo fondamentali:

Controllo della presenza della rete

```
1 public static Boolean checkConnection(Context context) {
2     ConnectivityManager manager = (ConnectivityManager) context.getSystemService(Context.
3         CONNECTIVITY_SERVICE);
4     Network network = manager.getActiveNetwork();
5     if (network == null) return false;
6     NetworkCapabilities capabilities = manager.getNetworkCapabilities(network);
7     return capabilities != null && (capabilities.hasTransport(NetworkCapabilities.TRANSPORT_WIFI) ||
    capabilities.hasTransport(NetworkCapabilities.TRANSPORT_CELLULAR));
8 }
```

📄 ../../client/app/src/main/java/com/infopoint/core/networking/NetworkManager.java

Recupero delle informazioni

```
1 public static ArrayList<ArtWork> retrieveArtwork(String username, String password, String token) {
2     ArrayList<ArtWork> result = new ArrayList<>();
3     try {
4         InetAddress address = InetAddress.getByName(Constants.SERVER_ADDR);
5         if (address.isReachable(2000)) { // Check if the server is reachable
6             Socket s = new Socket(address, Constants.SERVER_PORT);
7             // Read the response received from the server
8             BufferedReader reader = new BufferedReader(new InputStreamReader(s.getInputStream()));
9
10            // Wait until the server inform us that is ready to perform communication
11            while (s.getInputStream().available() < 1) {
12                Log.d(_TAG, "Waiting... ");
13            }
14
15            String serverReady = reader.readLine();
16            Log.d(_TAG, "Message: " + serverReady);
17
18            // Create the writer in order to write to the socket just opened
19            PrintWriter out = new PrintWriter(new BufferedWriter( new OutputStreamWriter(s.getOutp
20                utStream()))), true);
21
22            out.println(String.format("◇REQUEST:RETRIEVE◇USERNAME:%s◇PASSWORD:%s◇TOKEN:%s\n",
23                username, password, token));
24            out.println("\n");
25
26            String buffer;
27            while (true) {
28                buffer = reader.readLine();
29                buffer = buffer.replace("\u0000", "");
30                if (buffer.isEmpty() || buffer.equals("◇RESPONSE:SUCCESS◇")) {
31                    Log.d(_TAG, "Finished artwork collection");
32                    break;
33                }
34
35                if (buffer.equals("◇RESPONSE:Invalid token◇") || buffer.equals("◇RESPONSE:FAILED◇")
36                    )) {
37                    Log.d(_TAG, "An error as occurred while retrieving");
38                    return null;
39                }
40
41                // Log.d(_TAG, "Buffer: " + buffer);
42                String[] fields = buffer.split(Constants.OUTER_DELIMITER);
43                // Log.d(_TAG, "Fields: " + Arrays.toString(fields));
44                Log.d(_TAG, "Name: " + fields[1].replace("NAME:", "") +
45                    " Author: " + fields[2].replace("AUTHOR:", "") +
46                    " Date: " + fields[3].replace("DATE:", "") +
47                    " Description" + fields[4].replace("DESCRIPTION:", ""));
48
49                result.add(new ArtWork(fields[1].replace("NAME:", ""),
50                    fields[2].replace("AUTHOR:", ""),
51                    fields[3].replace("DATE:", ""),
52                    fields[4].replace("DESCRIPTION:", "")));
53            }
54
55            Log.d(_TAG, "Closing socket");
56            s.close();
57        }
58    }
59 }
```

```

55     } catch (IOException e){
56         Log.d(_TAG, "Cause: " + e.getLocalizedMessage());
57     }
58     return result;
59 }

```

📄 ../../client/app/src/main/java/com/infopoint/core/networking/NetworkManager.java

Operazioni per l'utente ¹²

```

1  public static boolean user_operation(String requestType, String username, String password, String
2      token, Context ctx) {
3      try {
4          InetAddress address = InetAddress.getByName(Constants.SERVER_ADDR);
5          if (address.isReachable(Constants.SERVER_TIMEOUT)) { // Check if the server is reachable
6              Socket s = new Socket(address, Constants.SERVER_PORT);
7
8              // Reader of the socket
9              BufferedReader reader = new BufferedReader(new InputStreamReader(s.getInputStream()));
10
11             // Wait until the server inform us that is ready to perform communication
12             while(s.getInputStream().available() < 1) {
13                 Log.d(_TAG, "Waiting... ");
14             }
15
16             String msg = reader.readLine();
17             Log.d(_TAG, "Message: " + msg);
18
19             // Create the writer in order to write to the socket just opened
20             PrintWriter out = new PrintWriter(new BufferedWriter( new OutputStreamWriter(s.getOutp
21                 utStream())), true);
22
23             // Control switch to select the actual request to perform
24             String actualRequestType = switch (requestType) {
25                 case "LOGIN" → "LOGIN";
26                 case "REGISTRATION" → "REGISTRATION";
27                 case "DELETE" → "DELETE";
28                 default → "";
29             };
30
31             out.println(String.format("◇REQUEST:%s◇USERNAME:%s◇PASSWORD:%s◇TOKEN:%s\n",
32                 actualRequestType, username, password, token));
33             out.println("\n");
34
35             String buffer = reader.readLine();
36             buffer = buffer.replace("\u0000", "");
37             String[] fields = buffer.split(Constants.OUTER_DELIMITER);
38             Log.d(_TAG, "[RESPONSE]: " + fields[1].replace("RESPONSE:", ""));
39
40             if (fields[1].replace("RESPONSE:", "").equals("SUCCESS")) {
41
42                 // Save credentials and then move to HomePage
43                 StorageManager.with(ctx).write(Constants.USERNAME, username);
44                 StorageManager.with(ctx).write(Constants.PASSWORD, password);
45                 StorageManager.with(ctx).write(Constants.TOKEN, token);
46             } else {
47                 return false;
48             }
49
50             if (actualRequestType.equals("REGISTRATION")) {
51                 buffer = reader.readLine();
52                 buffer = buffer.replace("\u0000", "");
53                 fields = buffer.split(Constants.OUTER_DELIMITER);
54                 // Log.d(_TAG, "[TOKEN]: " + fields[1].replace("TOKEN:", ""));
55
56                 if (!fields[1].replace("TOKEN:", "").isEmpty()){
57                     // Save credentials and then move to HomePage
58                     StorageManager.with(ctx).write(Constants.TOKEN, fields[1].replace("TOKEN:", ""));
59                 } else {
60                     return false;
61                 }
62             }
63         }
64     }
65 }


```

```

61         Log.d(_TAG, "Closing socket");
62         s.close();
63         return true;
64     }
65     } catch (IOException e){
66         Log.d(_TAG, "Cause: " + e.getLocalizedMessage());
67     }
68     return false;
69 }
70 }

```

▢ ../../client/app/src/main/java/com/infopoint/core/networking/NetworkManager.java

Vista la natura di queste richieste, tutte le operazioni vengono eseguite da un thread¹³. Di seguito riportiamo un esempio concreto del loro utilizzo, preso dalla classe  **LoginActivity**:

```

1  public class LoginActivity extends AppCompatActivity {
2      private void handleLogin(String username, String password, String token) {
3          if (!Validator.validate(username) || !Validator.validate(password)) {
4              Toasty.error(this, "Attenzione!\nI campi inseriti non rispettano i requisiti richiesti!",
5                  Toasty.LENGTH_LONG).show();
6              return;
7          }
8          if (token == null || token.isEmpty()) {
9              Toasty.error(this, "Attenzione!\nIl token di autenticazione non può essere nullo!", Toasty.
10                  LENGTH_LONG).show();
11              return;
12          }
13          Thread task = new Thread(() → {
14              if (NetworkManager.user_operation(_REQUEST_TYPE, username, password, token, this)) {
15                  Log.d(_TAG, "Successfull registration! Moving to HomePage... ");
16                  startActivity(new Intent(LoginActivity.this, MainActivity.class));
17                  finishAffinity();
18                  overridePendingTransition(R.anim.slide_in_left, R.anim.slide_out_right);
19              } else {
20                  runOnUiThread(() → Toasty.info(this, "Non sono state trovate buche nella zona intorno a
21                      te", Toast.LENGTH_LONG, true).show());
22              }
23          });
24          task.setPriority(10);
25          task.start();
26      }
27  }

```

▢ ../../client/app/src/main/java/com/infopoint/ui/activity/authentication/login/LoginActivity.java

¹³Cerchiamo di evitare a tutti i costi di effettuare operazioni bloccanti, rendendo in questo modo l'applicazione irresponsive per tutta la durata della comunicazione

CAPITOLO: CODICE SORGENTE SVILUPPATO

Il codice sorgente prodotto durante lo sviluppo di *InfoPoint*[®] è disponibile sulla piattaforma *GitHub*, che ne ha permesso anche il versionamento.

Di seguito riportiamo un link per il **download**¹

¹Potrebbe essere richiesta l'autenticazione (il repository è per privacy privato)

5

CAPITOLO: RINGRAZIAMENTI

Ringraziamo la professoressa Alessandra Rossi per lo splendido corso, che ci ha permesso di conoscere nuove interessanti tecnologie e del supporto offertoci durante e dopo le lezioni.



LUCIA BRANDO

 N86003382

 l.brand@studenti.unina.it

 **Profilo**



VALENTINO BOCCHETTI

 N86003405

 vale.bocchetti@studenti.unina.it

 **Profilo**

Profilo 

da.morace@studenti.unina.it 

N86003778 

DARIO MORACE

