Università degli Studi di Napoli
Federico II

A.A. 2022-2023

InfoPoint

Lucia Brando l.brando@studenti.unina.it N86003382

Dario Morace da.morace@studenti.unina.it N86003778

Valentino Bocchetti vale.bocchetti@studenti.unina.it N86003405

Indice

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, hostato[1] su una macchina virtuale offerta da Azure[2];
- Una applicazione `Android`, scritta in `Java` che fa da client;

[1] Indica 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

[2] Per maggiori informazioni visitare il seguente sito

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 piattaforma[1];
- Possibilità di usufruire dei contenuti in base alla tipologia di utente, in modo da permette un focus diverso in base alle sue caratteristiche[2] ;

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 monolitico[4].

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: epoll, send, recv, socket, nanosleep, threads, mutex, semaphores

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, la combinazione di questi tool ha permesso un passaggio immediato da una situazione di esecuzione locale (di debug) ad una

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

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 elasticità [9] e scalabilità[10] nel tempo.

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

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

[3] Metodologia 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.

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

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

[6] Per maggiori informazioni visitare il seguente sito

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

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

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

[10] Per 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, potendo scegliere tra 3 diverse metodologie:

- Sblocco mediante **username** e **password**;
- Sblocco con **impronta digitale** (se supportato);
- Sblocco con **riconoscimento facciale** (se supportato);

1.2.2 Post registrazione

In seguito ad una corretta registrazione l'utente accede alla HomePage, nella quale ha la possibilità di visualizzare:

- L'elenco delle **artwork** presenti nel museo;
- Il proprio profilo;
- 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;

1.2.4 Modelli di dominio

Class Diagram

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

Sequence Diagram

Di seguito riportiamo il diagramma delle classi di Analisi prodotto durante lo sviluppo della piattaforma InfoPoint, in merito alla:

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

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 dati[1] .

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 pratices** 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 client[2];
- Vengono effettuati controlli e gestione degli stati di tutte le operazioni lato **Server**;
- Tutti i dati scambiati tra **Server** e **Client** vengono opportunamento controllati [3], onde evitare operazioni che possano minare il corretto funzionamento dell'applicativo

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

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

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

3

Capitolo: Dettagli implementativi

Estratto

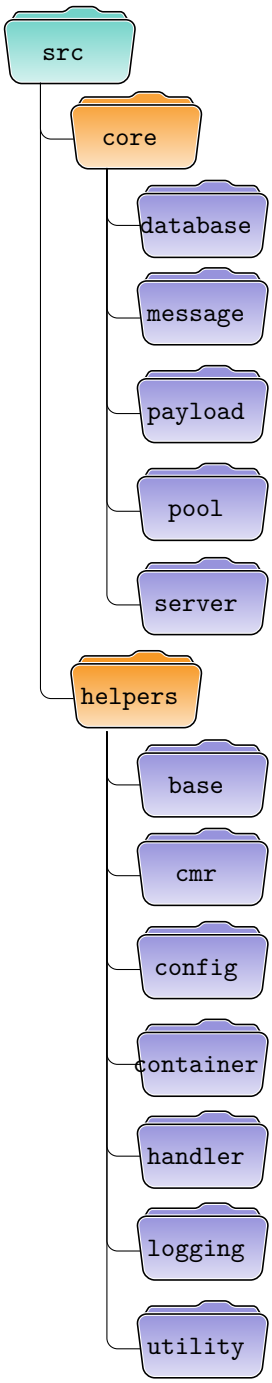
In questo capitolo tratteremo l'implementazione e il funzionamento delle componenti che hanno reso possibile lo sviluppo della piattaforma **InfoPoint**, prestando particolare attenzione alle funzionalità richieste da programma

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 socket[1];
- Analisi delle funzioni di logging e controllo degli stati utilizzate;
- Analisi e controllo delle principali system call per la gestione del Database utilizzato.

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.

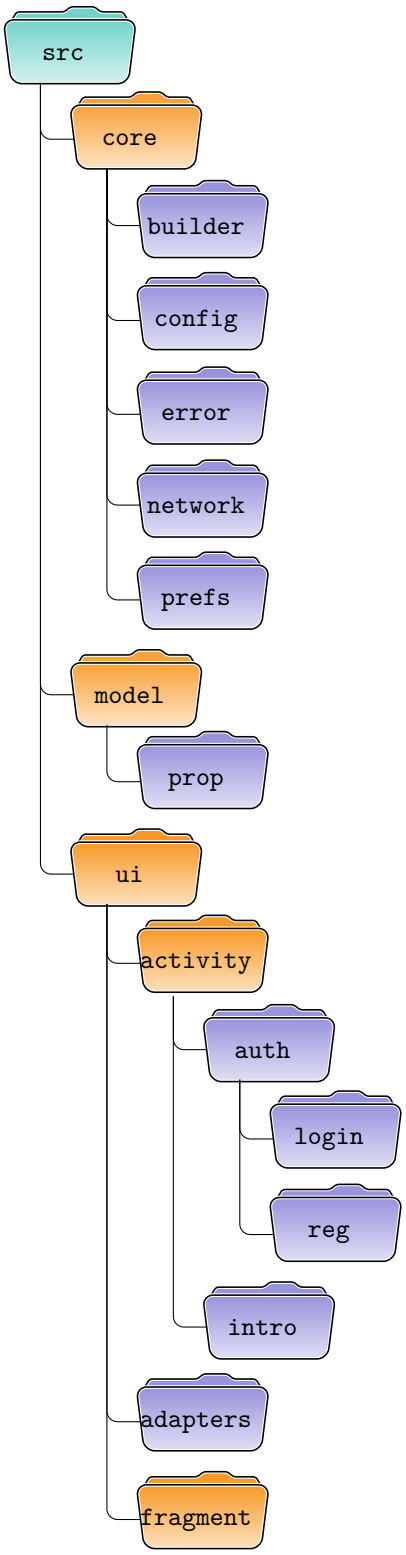
Helpers

(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 dispozione 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).

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

Di seguito riportiamo alcuni dettagli implementativi della struttura del Client.



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

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 activity [2] per
  - Autenticazione, ovvero
    - Login;
    - Registrazione (reg)
  - Esposizione dell'applicazione (Intro)
- **adapters** →Offre gli adapters [3] utilizzati dalle activity per la gestione delle componenti grafiche che fanno uso di componenti oggetto
- **fragments** →Contiene la definizione di tutti i fragment [4] presenti nell'applicazione

La "finestra" grafica con cui l'utente interagisce

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

La porzione "riusabile" della UI dell'applicazione

4 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]

[1] Potrebbe 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.brando@studenti.unina.it

Profilo

Valentino Bocchetti

N86003405

vale.bocchetti@studenti.unina.it

Profilo

Profilo

da.morace@studenti.unina.it

N86003778

Dario Morace