MyTalk

Sofware di comunicazione tra utenti senza requisiti di installazione

clockworkTeam7@gmail.com

Definizione di Prodotto

v 2.0

Informazioni sul documento

Nome documento	Definizione di Prodotto
Versione documento	v 2.0
Data creazione	2013/02/20
Data ultima modifica	2013/07/19
Uso documento	Esterno
Redazione	Zohouri Haghian Pardis Bain Giacomo
Verifica	La Bruna Agostino
Approvazione	Gavagnin Jessica
Lista distribuzione	gruppo *Clockwork* Zucchetti SPA Prof. Tullio Vardanega

Sommario

Il presente documento intende descrivere in modo dettagliato tutte le componenti del Progetto **MyTalk** e i criteri con le quali interagiscono fra loro. Per ogni componente sono illustrati gli attributi con il loro significato e i metodi con il loro comportamento.

Diario delle modifiche

Autore	Modifica	Data	Versione
Gavagnin Jessica	Approvazione del documento	2013/07/19	v 2.0
La Bruna Agostino	Verfica del documento	2013/07/18	v 1.8
Zohouri Haghian Pardis	Inseriti i metodi per effettuare la teleconferenza nelle classi SideView, ContactView, FunctionView, CallCommunication	2013/07/15	v 1.8
Ceseracciu Marco	Aggiunto il tracciamento requisiti-componenti	2013/07/16	v 1.6
Ceseracciu Marco	Aggiunto il tracciamento componenti-requisiti	2013/07/15	v 1.5
Zohouri Haghian Pardis	Modificate tutte le immagini dell'intero documento	2013/07/14	v 1.4
Zohouri Haghian Pardis	Sistemate classi DAO portandole in Singleton	2013/07/14	v 1.3
Zohouri Haghian Pardis	Inseriti parametri ed eccezioni lanciate dai metodi	2013/07/14	v 1.2
Zohouri Haghian Pardis	Correzioni tempi e struttura documento	2013/07/12	v 1.1
Bain Giacomo	Approvazione documento	2013/05/20	v 1.0
Furlan Valentino	Verifica generale del documento	2013/05/19	v 0.16
La Bruna Agostino	Verifica capitoli 4 e 5	2013/05/17	v 0.15
Zohouri Haghian Pardis	Verifica dal capitolo 1 al capitolo 3	2013/05/16	v 0.14
Gavagnin Jessica	Modifiche apportate al capitolo 3	2013/05/14	v 0.13
Ceseracciu Marco	Modifiche apportate al capitolo 4	2013/05/013	v 0.12
Gavagnin Jessica	Stesura capitolo 5	2013/04/06	v 0.11
Ceseracciu Marco	Stesura finale del capitolo 3	2013/04/06	v 0.10
Palmisano Maria Antonietta	Stesura finale del capitolo 4	2013/04/06	v 0.9
Ceseracciu Marco	Stesura del capitolo 4	2013/04/05	v 0.8
Gavagnin Jessica	Stesura del capitolo 3, package usermanager	2013/03/29	v 0.7
Gavagnin Jessica	Stesura del capitolo 3, package functionmanager	2013/03/27	v 0.6
Ceseracciu Marco	Stesura del capitolo 3, package view	2013/03/26	v 0.5
Gavagnin Jessica	Stesura del capitolo 3, package transfer	2013/03/21	v 0.4
Gavagnin Jessica	Stesura del capitolo 3, package usermanager	2013/03/20	v 0.3
Gavagnin Jessica	Creazione e stesura del capitolo 3, package shared e dao	2013/03/18	v 0.2

Palmisano Maria Antonietta	Creazione documento, stesura sezione Introduzione	2013/02/20	v 0.1

Indice

Elenco delle figure

Elenco delle tabelle

1 Introduzione

1.1 Scopo del documento

Il presente documento descrive tutte le componenti del sistema e il modo in cui esse collaborano. Per ogni componente vengono illustrati gli attributi con il loro significato e i metodi con il loro comportamento. Lo scopo principale del documento è quello di fornire ai programmatori una solida e precisa guida alla codifica, in modo che essi possano lavorare in modo autonomo attenendosi alle scelte progettuali ed evitando soluzioni personali ed improvvisate.

1.2 Scopo del prodotto

Il prodotto denominato **MyTalk** si propone di fornire un software per un sistema di comunicazione audio e video tra utenti. Lo scopo del progetto è poter comunicare con altri utenti tramite il browser, utilizzando solo componenti standard, senza dover installare plugin o programmi esterni. L'utilizzatore dovrà poter chiamare un altro utente, iniziare la comunicazione sia audio che video, svolgere la chiamata e terminare la chiamata ottenendo delle statistiche sull'attività.

1.3 Glossario

Per evitare ambiguità i termini tecnici o di uso non comune vengono evidenziati, alla loro prima occorrenza nel documento, tramite sottolineatura. Le definizioni di questi termini sono riportate nel documento in allegato `Glossario_v2.0.pdf`.

1.4 Riferimenti

1.4.1 Normativi

- Capitolato d'Appalto: **MyTalk**, rilasciato da Zucchetti SPA, reperibile all'indirizzo: `http://www.math.unipd.it/~tullio/IS-1/2012/Progetto/C1.pdf`
- Norme di Progetto (allegato `Norme_di_Progetto_v4.0.pdf`)

1.4.2 Informativi

- Analisi dei Requisiti (allegato `Analisi_dei_Requisiti_v4.0.pdf`)
- Specifica Tecnica (allegato `Specifica_Tecnica_v3.0.pdf`)

2 Standard di progetto

2.1 Standard di progettazione architetturale

Per gli standard di progettazione architetturale si veda il documento Specifica Tecnica (allegato `Specifica_Tecnica_v3.0.pdf`) e il documento Norme di Progetto (allegato `Norme_di_Progetto_v4.0.pdf`).

2.2 Standard di documentazione del codice

Per gli standard di documentazione del codice utilizzati si faccia riferimento al documento relativo alle Norme di Progetto (allegato `Norme_di_Progetto_v4.0.pdf`).

2.3 Standard di denominazione di entità e relazioni

Tutti gli elementi definiti, siano essi package, classi, metodi o attributi, devono avere denominazioni chiare ed autoesplicative, utilizzando nomi quanto più brevi possibili. Si faccia riferimento alle Norme di Progetto per ulteriori informazioni (allegato `Norme_di_Progetto_v4.0.pdf`).

2.4 Standard di programmazione

Gli standard di programmazione sono stati definiti nel documento Norme di Progetto. Si rimanda ad esso per l'elenco delle regole da seguire in fase di scrittura del codice (allegato `Norme_di_Progetto_v4.0.pdf`).

2.5 Strumenti di lavoro

Gli strumenti utilizzabili per la realizzazione dei prodotti sono tutti e soli quelli stabiliti nelle Norme di Progetto (allegato `Norme_di_Progetto_v4.0.pdf`).

3 Comunicazione Client-Server

Comunicazione tra AuthenticationComunication e AuthenticationTransfer

Autenticazione:
AuthenticationComunication invia a AuthenticationTransfer una stringa avente:

- **type**: *login*
- **altri attributi:** username, password

AuthenticationTransfer risponde inviando a AuthenticationComunication una stringa avente:

- **type**: *login*
- **altri attributi:** answer, ((name, surname) | error)
  - Descrizione attributi: se answer è *true* nel pacchetto vengono inseriti il nome e il cognome relativi al dato username, presenti nella base di dati. Altrimenti viene inserito error che può assumere come valori: *Password errata* o *Username errato*

Registrazione:
AuthenticationComunication invia a AuthenticationTransfer una stringa avente:

- **type**: *signUp*
- **altri attributi:** username, password, name, surname

AuthenticationTransfer risponde inviando a AuthenticationComunication una stringa avente:

- **type**: *signUp*
- **altri attributi:** answer, (error)
  - Descrizione attributi: se answer risulta *false* viene inserito error che può assumere come valori: *Errore nell'inserimento dell'utente nel database* o *Username già presente*

Disconnessione:
AuthenticationComunication invia a AuthenticationTransfer una stringa avente:

- **type**: *logout*

Comunicazione tra CallComunication e CallTransfer

Instaurazione di una chiamata singola:
CallComunication invia a CallTransfer una stringa avente:

- **type**: *call*
- **altri attributi:** contact, callType, conference
  - Descrizione degli attributi: callType conterrà il tipo di chiamata, *video* o *audio*, conference conterrà *false*

Instaurazione di una conferenza:
CallComunication invia a CallTransfer una stringa avente:

- **type**: *call*
- **altri attributi:** contact, callType, conference
  - Descrizione degli attributi: callType conterrà il tipo di chiamata, *video* o *audio*, conference conterrà *true*

Accettazione di una chiamata:
CallComunication invia a CallTransfer una stringa avente:

- **type**: *answeredCall*
- **altri attributi:** contact, conference
  - Descrizione degli attributi: conference conterrà *false* se si tratta di una chiamata singola, *true* se si tratta di una conferenza

CallTransfer risponde inviando a CallComunication una stringa avente:

- **type**: *answeredCall*
- **altri attributi:** answer, user
  - Descrizione degli attributi: answer contiene la stringa *true,* user rappresenta l'utente che ha accettato la chiamata

Negazione di una chiamata:
CallTransfer invia a CallComunication una stringa avente:

- **type**: *answeredCall*
- **altri attributi:** answer, error
  - Descrizione degli attributi: answer contiene la stringa *false,* error contiene il motivo di rifiuto, scelto tra *Utente non connesso al server, Chiamata rifiutata, Utente occupato in un'altra conversazione, Utente rifiuta di accendere la telecamera*

Segnalazione presenza contatti videoconferenza
CallCommunication invia a CallTransfer una stringa avente:

- **type**: *addConferenceCaller*
- **altri attributi:** contact, user
  - Descrizione attributi: user contiene la stringa di un nuovo utente aggiuntosi alla conferenza

CallTransfer risponde inviando a CallCommunication una stringa avente:

- **type**: *addConferenceCaller*
- **altri attributi:** user

CallCommunication invia a CallTransfer una stringa avente:

- **type**: *addConferenceAnswer*
- **altri attributi:** contact, user

CallTransfer risponde inviando a CallCommunication una stringa avente:

- **type**: *addConferenceAnswer*
- **altri attributi:** user

Protocollo della descrizione della sessione:
CallComunication invia a CallTransfer una stringa avente:

- **type**: *sdp*
- **altri attributi:** contact, description

CallTransfer risponde inviando a CallComunication una stringa avente:

- **type**: *offer*
- **altri attributi:** Stringa creata da WebRTC, contact
  - Descrizione degli attributi: la stringa non è altro che la stringa contenuta nel attributo description passato precedentemente

CallComunication risponde inviando a CallTransfer una stringa avente:

- **type**: *sdp*
- **altri attributi:** contact, description
  - NOTA: la stringa contenuta in description è diversa da quella inviata nel primo passaggio, infatti mentre nel primo caso la stringa proveniva dal chiamante, in questo caso proviene dal chiamato

CallTransfer risponde inviando a CallComunication una stringa avente:

- **type**: *answer*
- **altri attributi:** contact,description
  - NOTA: la stringa non è altro che la stringa contenuta nel attributo description passato precedentemente

Invio dei candidati per la configurazione della chiamata
CallComunication invia a CallTransfer una stringa avente:

- **type**: *candidateReady*
- **altri attributi:** contact

CallTransfer risponde inviando a CallComunication una stringa avente:

- **type**: *candidateReady*
- **altri attributi:** contact

CallComunication risponde inviando a CallTransfer una stringa avente:

- **type**: *candidate*
- **altri attributi:** contact, candidate

CallTransfer risponde inviando a CallComunication una stringa avente:

- **type**: *candidate*
- **altri attributi:** contact
  - NOTA: la stringa non è altro che la stringa contenuta nel attributo candidate passato precedentemente

Terminazione di una chiamata:
CallComunication invia a CallTransfer una stringa avente:

- **type**: *endCall*
- **altri attributi:** contact

CallTransfer risponde inviando a CallComunication una stringa avente:

- **type**: *endCall*
- **altri attributi:** contact

Terminazione di una chiamata anticipata:
CallComunication invia a CallTransfer una stringa avente:

- **type**: *endCallEarly*
- **altri attributi:** contact

CallTransfer risponde inviando a CallComunication una stringa avente:

- **type**: *endCallEarly*
- **altri attributi:** contact

Comunicazione tra ChatComunication e ChatTransfer

Invio riuscito di un messaggio:
ChatComunication invia a ChatTransfer una stringa avente:

- **type**: *sendText*
- **altri attributi:** contact, message

ChatTransfer risponde inviando a ChatComunication una stringa avente:

- **type**: *sendText*
- **altri attributi:** message, contact

Invio non riuscito di un messaggio:
ChatTransfer risponde inviando a ChatComunication una stringa avente:

- **type**: *notDelivered*
- **altri attributi:** message, contact

Comunicazione tra ContactsComunication e AuthenticationTransfer

Richiesta degli utenti presenti nel server:
ContactsComunication invia a AuthenticationTransfer una stringa avente:

- **type**: *getContacts*

AuthenticationTransfer risponde inviando a ContactsComunication una stringa avente:

- **type**: *getContacts*
- **altri attributi:** size, username$i$, name$i$, surname$i$, *IP*i
  - Descrizione degli attributi: username$i$, name$i$, surname$i$, IPi vengono ripetuti per il numero di utenti presenti nel server, i quindi parte da 0 e arriva a size-1

Nota: AuthenticationTransfer invia a ContactsComunication la stringa anche nei casi di ricezione di stringe *login, signUp, logout* dalla classe AuthenticationCommunication e *changeData* dalla classe UserDataCommunication. In questi casi tuttavia non invia tutta la lista di utenti ma un solo utente.

Comunicazione tra FileComunication e FileTransfer

Invio di un file:
FileComunication invia a FileTransfer una stringa avente:

- **type**: *file*
- **altri attributi:** file, contact

Avviso del rifiuto di un file da parte del destinatario:
FileTransfer invia a FileComunication una stringa avente:

- **type**: *fileRefused*
- **altri attributi:** contact, error
  - Descrizione attributi: error conterrà il motivo per la mancata consegna del file, e quindi *L'utente non risulta connesso al server* o *L'utente ha rifiutato il file*

Comunicazione tra NotificationComunication e CallTransfer

Avviso di chiamata:
CallTransfer invia a NotificationComunication una stringa avente:

- **type**: *call*
- **altri attributi:** contact, callType

Avviso di utente chiamato occupato:
NotificationComunication invia a CallTransfer una stringa avente:

- **type**: *busy*
- **altri attributi:** contact

Rifiuto di una chiamata:
NotificationComunication invia a CallTransfer una stringa avente:

- **type**: *refuseCall*
- **altri attributi:** contact

Avviso di annullamento di una chiamata, quindi in fase di inizializzazione:
CallTransfer invia a NotificationComunication una stringa avente:

- **type**: *endCallEarly*
- **altri attributi:** contact

Comunicazione tra NotificationComunication e FileTransfer

Avviso di ricezione di un file:
FileTransfer invia a NotificationComunication una stringa avente:

- **type**: *file*
- **altri attributi:** file, contact

Rifiuto di un file:
NotificationComunication invia a FileTransfer una stringa avente:

- **type**: *refuseFile*
- **altri attributi:** contact

Comunicazione tra NotificationComunication e RecordMessageTransfer

Invio dei messaggi audio/video all'utente:
RecordMessageTransfer invia a NotificationComunication una stringa avente:

- **type**: *getRecords*
- **altri attributi:** size, sender$i$, message$i$, dateCreation$i$
  - Descrizione degli attributi: sender$i$, message$i$, dateCreation$i$ vengono ripetuti per il numero di messaggi, inviati all'utente, presenti nel server, i quindi parte da 0 e arriva a size-1

Nota: RecordMessageTransfer potrebbe inviare a NotificationComunication la stringa anche nei casi di ricezione della stringa *sendRecord* dalla classe RecordMessageCommunication. In questi casi tuttavia invia il singolo messaggio ricevuto.

Comunicazione tra RecordMessageComunication e RecordMessageTransfer

Invio di un messaggio audio/video:
RecordMessageComunication invia a RecordMessageTransfer una stringa avente:

- **type**: *sendRecord*
- **altri attributi:** contact, path, date

RecordMessageTransfer risponde inviando a RecordMessageComunication una stringa avente:

- **type**: *sendRecord*
- **altri attributi:** answer, (error)
  - Descrizione degli attributi: nel caso in cui answer contenga *false*, error segnalerà il motivo del mancato invio del messaggio, e quindi conterrà la stringa *Errore nella registrazione del messaggio*

Eliminazione di un messaggio audio/video:
RecordMessageComunication invia a RecordMessageTransfer una stringa avente:

- **type**: *removeRecord*
- **altri attributi:** contact, path, date

RecordMessageTransfer risponde inviando a RecordMessageComunication una stringa avente:

- **type**: *removeRecord*
- **altri attributi:** answer, (error)
  - Descrizione degli attributi: nel caso in cui answer contenga *false*, error segnalerà il motivo del mancato invio del messaggio, e quindi conterrà la stringa *Errore nella cancellazione del messaggio*

Comunicazione tra TutorialComunication e Authentication-Transfer

Rifiuto di un file:
AuthenticationTransfer invia a TutorialComunication una stringa avente:

- **type**: *tutorials*
- **altri attributi:** size, title*i*, path*i*
  - Descrizione degli attributi: title*i*, path*i* vengono ripetuti per il numero di tutorial presenti nel server, *i* quindi parte da 0 e arriva a size-1

Comunicazione tra UserDataComunication e UserTransfer

Controllo credenziali:
UserDataCommunication invia a UserTransfer una stringa avente:

- **type**: *checkCredentials*
- **altri attributi:** password

UserTransfer risponde inviando a UserDataCommunication una stringa avente:

- **type**: *checkCredentials*
- **altri attributi:** answer

Cambio dati, credenziali corrette:
UserDataCommunication invia a UserTransfer una stringa avente:

- **type**: *changeData*
- **altri attributi:** name, surname, password

UserTransfer risponde inviando a UserDataCommunication una stringa avente:

- **type**: *changeData*
- **altri attributi:** answer, (error)
  - Descrizione degli attributi: se answer contiene il valore *false*, allora error conterrà il tipo di errore, cioè una stringa tra *Errore nell'operazione di modifica del nome e del cognome, Errore nell'operazione di modifica della password* o *Username errato*

4 Specifica Client

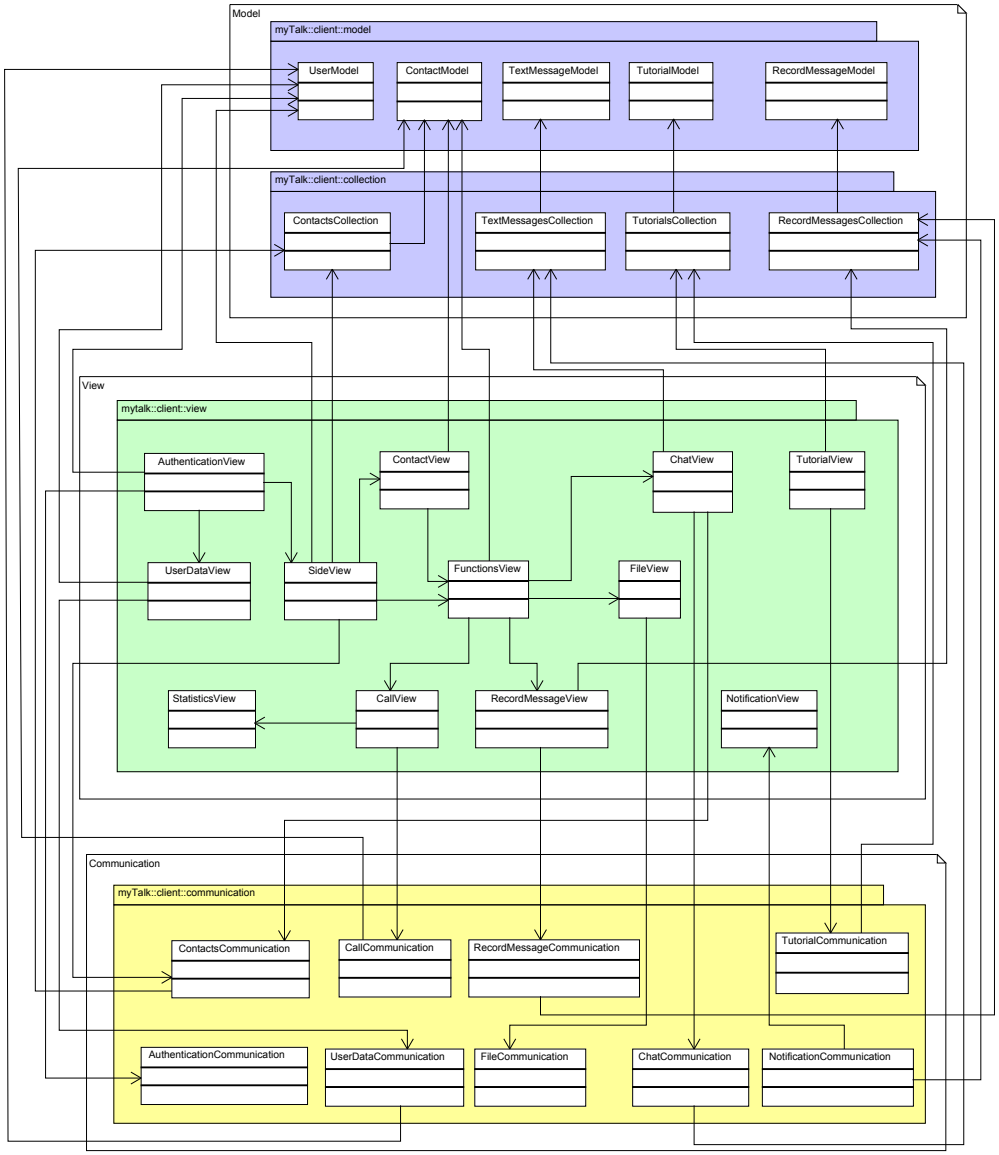

Figura 1: Architettura del client

Il client contiene i seguenti package:

- `mytalk.client.view` (vedasi sezione 4.1)
- `mytalk.client.communication` (vedasi sezione 4.2)
- `mytalk.client.collection` (vedasi sezione 4.3)
- `mytalk.client.model` (vedasi sezione 4.4)

4.1 Package view

Il package view costituisce la parte del sistema che definisce ed implementa l'interfaccia web usufruibile dagli utenti mediante pagine web. Diversamente da uno schema di tipo MVC il package view contiene non solo le parti di interfaccia ma anche parti logiche, come previsto dal framework Backbone.js, che prevede uno schema di tipo MV*.

Costituito dalle classi:

- `mytalk.client.view.AuthenticationView` (vedasi sezione 4.1.1)
- `mytalk.client.view.CallView` (vedasi sezione 4.1.2)
- `mytalk.client.view.ChatView` (vedasi sezione 4.1.3)
- `mytalk.client.view.ContactView` (vedasi sezione 4.1.4)
- `mytalk.client.view.FileView` (vedasi sezione 4.1.5)
- `mytalk.client.view.FunctionsView` (vedasi sezione 4.1.6)
- `mytalk.client.view.NotificationView` (vedasi sezione 4.1.7)
- `mytalk.client.view.RecordMessageView` (vedasi sezione 4.1.8)
- `mytalk.client.view.SideView` (vedasi sezione 4.1.9)
- `mytalk.client.view.StatisticsView` (vedasi sezione 4.1.10)
- `mytalk.client.view.TutorialView` (vedasi sezione 4.1.11)
- `mytalk.client.view.UserDataView` (vedasi sezione 4.1.12)

4.1.1 `mytalk.client.view.AuthenticationView`

La classe AuthenticationView deve definire la struttura, e la conseguente visualizzazione, della parte della pagina web che consente di effettuare la registrazione, l'accesso e la disconnessione dal sistema.

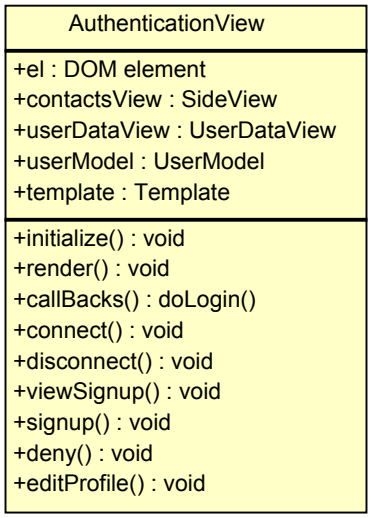

Figura 2: Classe AuthenticationView

Attributi pubblici:

- **[DOM element] el:** elemento DOM al quale è legata la classe
- [**SideView**] **contactsView:** riferimento alla classe SideView
- **[UserDataView] userDataView:** riferimento alla classe UserDataView
- **[UserModel] userModel:** riferimento alla classe UserModel
- **[Template] template:** riferimento al template utilizzato

Eventi:

- **click button#login:** evento che richiama la funzione *connect*
- **click button#logout:** evento che richiama la funzione *disconnect*
- **click button#signup:** evento che richiama la funzione *viewSignup*
- **click button#sign:** evento che richiama la funzione *signup*
- **click button#deny:** evento che richiama la funzione *deny*
- **click button#edit:**evento che richiama la funzione *editProfile*

Metodi pubblici:

- **[void] initialize():** metodo per inizializzare l'oggetto, si occupa di inizializzare *contactsView* e richiamare il metodo *render*
- **[void] render():** metodo per effettuare la scrittura nella pagina del template AuthenticationTemplate
- **[doLogin()] callBacks():** metodo che permette ad altre classi di usufruire del metodo *doLogin*
  - **[void] doLogin([String] username, [String] password, [String[]] result, [AuthenticationView] view)**: metodo che effettua l'operazione di login. Si occupa di creare il model corrispondente all'utente, aggiornare il template e invocare il metodo *getContacts* sull'oggetto *contactsView*
- **[void] connect():** metodo che avvia l'operazione di login dell'utente richiamando il metodo *checkCredentials* della classe AuthenticationCommunication passandogli come parametri lo username e la password inseriti negli appositi campi
- **[void] disconnect():** metodo che si occupa di chiudere la sessione con il server, invocando il metodo *logout* della classe AuthenticationCommunication, aggiornando il template e deallocando *userDataView* e *contactsView*
- **[void] viewSignup():** metodo che permette la visualizzazione del form di registrazione
- **[void] signup():** metodo che avvia l'operazione di registrazione dell'utente. Controlla che i campi obbligatori siano compilati, che lo username includa solo i caratteri permessi e che la password e quella di conferma coincidano. In caso affermativo invoca il metodo *signup* della classe AuthenticationCommunication, altrimenti mostra uno tra i messaggi di errore *Username può contenere solo lettere, numeri e underscore*, *Le password inserite non coincidono* e *Non hai compilato tutti i campi obbligatori*.
- **[void] deny():** metodo per annullare la compilazione della registrazione e tornare allo stato precedente
- **[void] editProfile()**: metodo che avvia l'operazione di modifica dati, inizializzando *userDataView*. Prima di effettuare l'operazione controlla che tutte le view presenti nel content siano state chiuse

4.1.2 `mytalk.client.view.CallView`

La classe CallView deve definire la struttura, e la conseguente visualizzazione, della pagina di chiamata

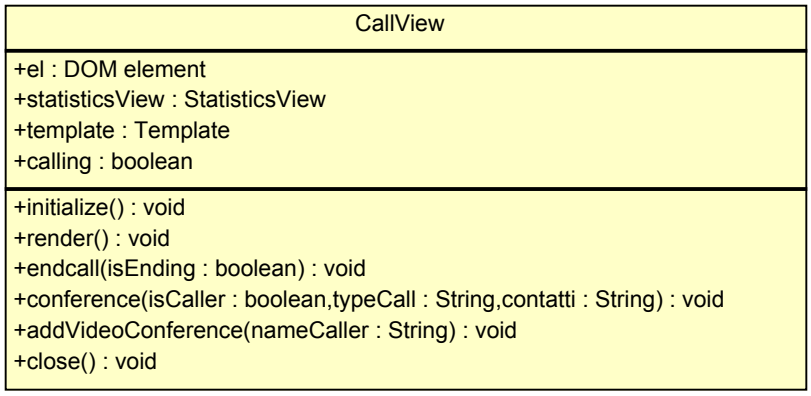

Figura 3: Classe CallView

Attributi pubblici:

- **[DOM element] el:** elemento DOM al quale è legata la classe
- **[StatisticsView] statisticsView:** riferimento alla classe StatisticsView
- **[Template] template:** riferimento al template utilizzato
- **[boolean] calling:** variabile che indica se l'utente è occupato in una chiamata o no

Eventi:

- **click button#endcall:** evento che richiama la funzione *endcall()*

Metodi pubblici:

- **[void] initialize():** metodo per inizializzare l'oggetto, si occupa di inizializzare *statisticsView* e richiamare il metodo *render*
- **[void] render():** metodo per effettuare la scrittura nella pagina del template CallTemplate e che verifica che sia in corso una chiamata e, in caso affermativo, invoca il metodo *recoverCall* di CallCommunication e il metodo *render* di StatisticsView. Altrimenti, a seconda che l'utente sia il chiamato o il chiamante, invoca i metodi *sendAnswer* o *sendCall* di CallCommunication e assegna a *calling* lo stato di chiamata
- **[void] endcall([boolean] isEnding):** metodo che termina la chiamata in corso richiamando i metodi *closeViewCall()* della classe FunctionsView, *close()* della classe stessa e *close()* della classe StatisticsView.

Inoltre nel caso il metodo sia invocato dall'utente che ha deciso di chiudere la chiamata, viene invocato anche il metodo *endCall()* della classe CallCommunication

- **[void] conference([boolean]isCaller, [String] typeCall, [String] contatti):** metodo che abilita gli eventi invocando il metodo *delegateEvents()* della classe stessa, aggiorna il template e verifica se è in corso una conferenza e, in caso affermativo, invoca il metodo *recoverCall* di CallCommunication. Altrimenti, a seconda che l'utente sia il chiamato o il chiamante, invoca i metodi *sendAnswer* o *sendCall* di CallCommunication e assegna a *calling* lo stato di chiamata
- **[void] addVideoConference([String] nameCaller):** metodo che aggiunge uno stream remoto ad una videoconferenza
- **[void] close():** metodo che invoca il metodo *endCall* di CallCommunication e chiude l'istanza della classe creata

4.1.3 `mytalk.client.view.ChatView`

La classe ChatView definisce la struttura, e la conseguente visualizzazione, della parte della pagina web che consente di effettuare comunicazioni testuali

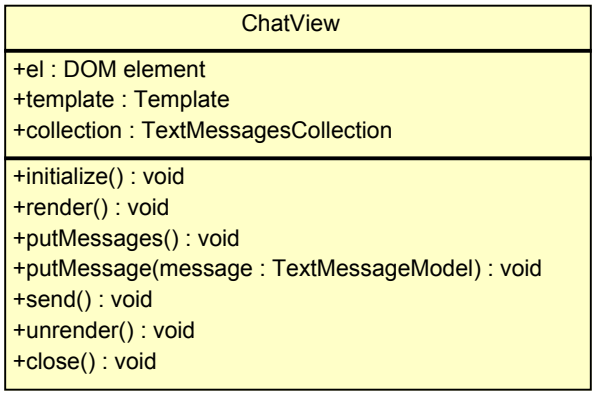

Figura 4: Classe ChatView

Attributi pubblici:

- **[DOM element] el:** elemento DOM al quale è legata la classe
- **[Template] template:** riferimento al template utilizzato

- **[TextMessagesCollection] collection:** riferimento alla classe TextMessagesCollection

Eventi:

- **click button#send:** evento che richiama la funzione *send()*

Metodi pubblici:

- **[void] initialize():** metodo per inizializzare l'oggetto, collega il metodo *render* della classe alla collection in modo da richiamarlo ogni qual volta avvenga un cambiamento di quest'ultima
- **[void] render():** metodo per effettuare la scrittura nella pagina del template ChatTemplate inoltre, invoca il metodo *putMessages* per visualizzare i messaggi scambiati fino ad ora
- **[void] putMessages()**: metodo che si occupa di prelevare dalla *collection* tutti i messaggi di testo della conversazione scelta, e invoca la funzione *putMessage* per permetterne la visualizzazione all'utente
- **[void] putMessage([TextMessageModel] message)**: metodo che si occupa della scrittura di un messaggio di testo, presente in *collection,* all'interno della pagina
- **[void] send():** metodo per inviare un messaggio al contatto, per fare ciò invoca il metodo *send* della classe ChatCommunication, con parametri lo username del contatto ed il testo del messaggio. Provvede anche ad aggiungere il messaggio a *collection*
- **[void] unrender():** metodo per chiudere l'istanza, per fare ciò rimuove da *collection* la conversazione con l'utente e invoca il metodo *close*
- **[void] close():** metodo che chiude l'istanza della classe creata

4.1.4 `mytalk.client.view.ContactView`

La classe ContactView consente la visualizzazione all'interno della pagina web di un singolo utente registrato

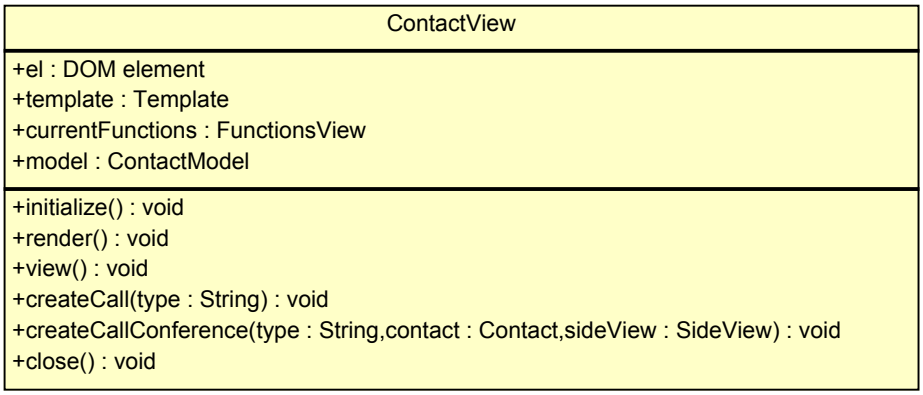

Figura 5: Classe ContactView

Attributi pubblici:

- **[DOM element] el:** elemento DOM al quale è legata la classe
- **[Template] template:** riferimento al template utilizzato
- **[FunctionsView] currentFunctions:** riferimento alla classe FunctionsView
- **[ContactModel] model:** riferimento al model del contatto visualizzato

Eventi:

- **click span.contact:** evento che richiama la funzione *view()*

Metodi pubblici:

- **[void] initialize():** metodo per inizializzare l'oggetto, collega il metodo *render* della classe al model in modo da richiamarlo ogni qual volta avvenga un cambiamento di quest'ultimo
- **[void] render():** metodo per effettuare la scrittura nella pagina del template ContactTemplate
- **[void] view():** metodo che si occupa di avviare la FunctionsView relativa al contatto. Prima di fare ciò invoca il metodo *closeOtherContacts* della classe SideView per controllare che tutte le view presenti nel content siano state chiuse

- **[void] createCall([String] type):** metodo che viene chiamato quando si accetta una chiamata in arrivo; invoca il metodo *closeOtherContacts* della classe SideView, per controllare che tutte le view presenti nel content siano state chiuse, e uno tra i metodi *videocall* o *audiocall* della classe FunctionsView, a seconda del tipo di chiamata passato dal parametro
- **[void] createCallConference([String] type, [Contact] contact, [SideView] sideView):** metodo che viene chiamato quando si accetta una richiesta di videoconferenza che invoca il metodo *closeOtherContacts* della classe SideView, per controllare che tutte le view presenti nel content siano state chiuse e invoca il metodo *conference* della classe FunctionsView
- **[void] close():** metodo che chiude l'istanza della classe creata invocando il metodo *close* della classe FunctionsView

4.1.5 `mytalk.client.view.FileView`

La classe FileView definisce la struttura, e la conseguente visualizzazione, della parte della pagina web che consente di selezionare un file salvato in locale per mandarlo all'utente autenticato selezionato.
Pur essendo pianificata tale classe non è stata implementata.

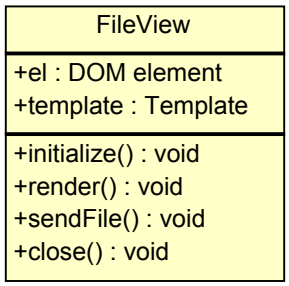

Figura 6: Classe FileView

Attributi pubblici:

- **[DOM element] el:** elemento DOM al quale è legata la classe
- **[Template] template:** riferimento al template utilizzato

Eventi:

- **click button#sendFile:** evento che richiama la funzione *sendFile*

Metodi pubblici:

- **[void] initialize():** metodo per inizializzare l'oggetto, si occupa di richiamare il metodo *render*
- **[void] render():** metodo per effettuare la scrittura nella pagina del template FileTemplate
- **[void] sendFile():** metodo che invia il file al server, attraverso l'utilizzo del metodo *sendFile* di FileCommunication
- **[void] close():** metodo che chiude l'istanza della classe creata

4.1.6 `mytalk.client.view.FunctionsView`

La classe FunctionsView definisce la struttura, e la conseguente visualizzazione, delle funzionalità offerte dopo aver selezionato un contatto

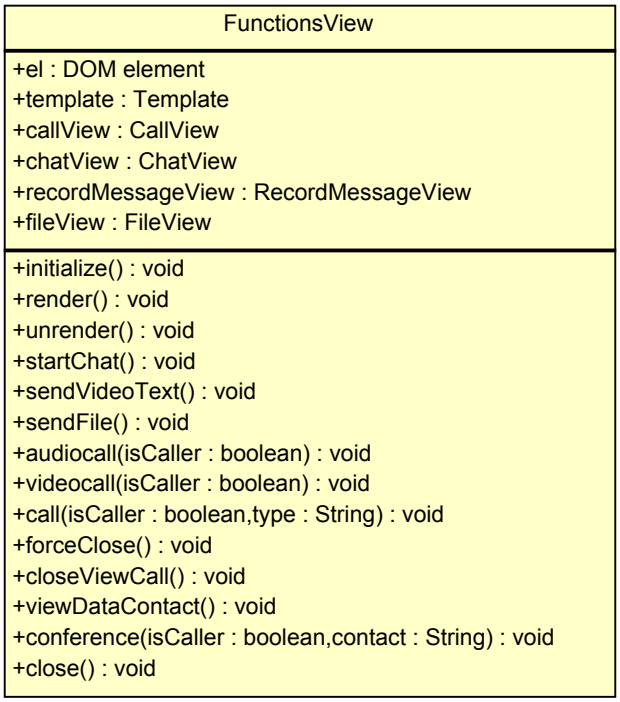

Figura 7: Classe FunctionsView

Attributi pubblici:

- **[DOM element] el:** elemento DOM al quale è legata la classe
- **[Template] template:** riferimento al template utilizzato
- **[CallView] callView:** riferimento alla classe CallView
- **[ChatView] chatView:** riferimento alla classe ChatView
- **[RecordMessageView] recordMessageView:** riferimento alla classe RecordMessageView
- **[FileView] fileView:** riferimento alla classe FileView

Eventi:

- **click button#dataContact:** evento che richiama la funzione *viewDataContact*
- **click button#sendVideoText:** evento che richiama la funzione *sendVideoText*
- **click button#sendFile:** evento che richiama la funzione *sendFile*
- **click button#call:** evento che richiama la funzione *audiocall*
- **click button#video:** evento che richiama la funzione *videocall*
- **click button#record:** evento che richiama la funzione *record*
- **click button#startConference:** evento che richiama la funzione *conference*

Metodi pubblici:

- **[void] initialize():** metodo per inizializzare l'oggetto, collega il metodo *render* della classe al model in modo da richiamarlo ogni qual volta avvenga un cambiamento di quest'ultimo
- **[void] render():** metodo per effettuare la scrittura nella pagina del template FunctionsTemplate, per fare ciò controlla se l'istanza è stata creata da SideView o da ContactView e se esiste già una chiamata in corso, nel qual caso mostra la chiamata
- **[void] unrender():** metodo per chiudere definitivamente l'istanza, invoca il metodo *unrender()* della classe ChatView e il metodo *close()* della classe stessa
- **[void] startChat():** metodo per avviare una comunicazione testuale con il contatto, per fare ciò inizializza *chatView*, se non è già esistente, e invoca il metodo *render* della classe ChatView

- **[void] sendVideoText():** metodo per inizializzare la vista per la registrazione di un messaggio, per fare ciò inizializza *recordMessageView*, se non è già esistente, e invoca il metodo *render()* della classe RecordMessageView
- **[void] sendFile():** metodo per inizializzare la vista per l'invio di un file, per fare ciò inizializza *fileView*, se non è già esistente, e invoca il metodo *render()* della classe fileView
- **[void] audiocall([boolean] isCaller):** metodo per avviare una chiamata audio, per fare ciò invoca il metodo *call(boolean, String),* passando come parametri quello di audiocall e la stringa 'audio'
- **[void] videocall([boolean] isCaller):** metodo per avviare una chiamata video, per fare ciò invoca il metodo *call(boolean, String),* passando come parametri quello di videocall e la stringa 'video'
- **[void] call([boolean] isCaller, [String] type):** metodo per gestire l'inizializzazione di una chiamata, per fare ciò inizializza *callView* e invoca il metodo *render(boolean, String, ContactModel, boolean)* specificando se si è il chiamante, il tipo di chiamata, il contatto con cui si sta comunicando e se si vuole registrare la chiamata. Inoltre si occupa di mantenere la chat attiva, richiamando il metodo *startChat()*
- **[void] forceClose():** metodo che forza la chiusura di una chiamata invocando *endcall()* della classe CallView
- **[void] closeViewCall():** metodo che, a chiamata terminata, ripristina la vista del contatto con cui si era in chiamata
- **[void] viewDataContact():** metodo che mostra i dati dell'utente selezionato
- **[void] conference([boolean] isCaller, [String] contact):** metodo che inizializza la conferenza. Controlla i contatti selezionati per la conferenza e li contatta. Si chiudono le chiamate presenti tramite invocato il metodo *forceClose()* della classe callView ed infine chiama il metodo *conference* di CallView verificando se si è o no il chiamante e passando i parametri. Altrimenti ripristina la conferenza
- **[void] close():** metodo che chiude l'istanza della classe creata e invoca il metodo *unrender()* delle classi ChatView e CallView

4.1.7 `mytalk.client.view.NotificationView`

La classe NotificationView definisce la struttura, e la conseguente visualizzazione, della parte della pagina per la segnalazione di notifiche

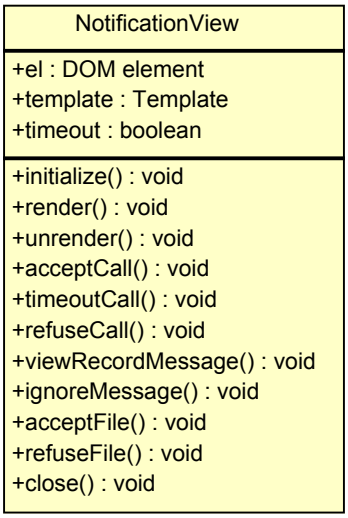

Figura 8: Classe NotificationView

Attributi pubblici:

- **[DOM element] el:** elemento DOM al quale è legata la classe
- **[Template] template:** riferimento al template utilizzato
- **[boolean] timeout:** variabile booleana che segnala quando non si risponde in tempo ad una notifica

Eventi:

- **click button#acceptCall:** evento che richiama la funzione *acceptCall*
- **click button#refuseCall:** evento che richiama la funzione *refuseCall*
- **click button#viewMessage:** evento che richiama la funzione *viewRecordMessage*
- **click button#ignoreMessage:** evento che richiama la funzione *ignoreMessage*
- **click button#acceptFile**: evento che richiama la funzione *acceptFile*
- **click button#refuseFile**: evento che richiama la funzione *refuseFile*

Metodi pubblici:

- **[void] initialize():** metodo per inizializzare l'oggetto, si occupa di inizializzare la variabile *timeout* a *true* e richiamare il metodo *render*
- **[void] render():** metodo per effettuare la scrittura nella pagina del template NotificationTemplate
- **[void] unrender():** metodo per rimuovere la vista delle notifiche, invoca il metodo *close()* della classe stessa
- **[void] acceptCall():** metodo per accettare una chiamata in arrivo, invoca il metodo *unrender* e assegna a *timeout* il valore *false,* infine se la chiamata avviata non era conferenza invia un evento di tipo *acceptCall* per segnalare che si sta iniziando una chiamata, altrimenti invia un evento di tipo *acceptCallConference*
- **[void] timeoutCall():** metodo per rifiutare una chiamata nel caso in cui non si risponda in tempo alla notifica, quindi se *timeout* risulta *true* chiama il metodo *refuseCall*
- **[void] refuseCall():** metodo per rifiutare una chiamata, per fare ciò invoca il metodo *refuse* della classe NotificationCommunication e il metodo *unrender* della classe stessa
- **[void] viewRecordMessage():** metodo per visualizzare il messaggio video/audio ricevuto, invoca il metodo *unrender* e invia un evento di tipo *viewMessage* che viene catturato da SideView
- **[void] ignoreMessage():** metodo per ignorare la visualizzazione del messaggio video/audio ricevuto, invoca il metodo *unrender*
- **[void] acceptFile():** metodo per accettare la ricezione di un file in arrivo, scarica il file e invoca il metodo *unrender*
- **[void] refuseFile():** metodo per rifiutare la ricezione di un file in arrivo, per fare ciò invoca il metodo *refuseFile* della classe NotificationCommunication e il metodo *unrender* della classe stessa
- **[void] close():** metodo che chiude l'istanza della classe creata

4.1.8 `mytalk.client.view.RecordMessageView`

La classe RecordMessageView definisce la struttura, e la conseguente visualizzazione, della parte della pagina web per effettuare registrazioni audio o audio/video da inviare ad un utente registrato.
Pur essendo pianificata questa classe non è stata implementata

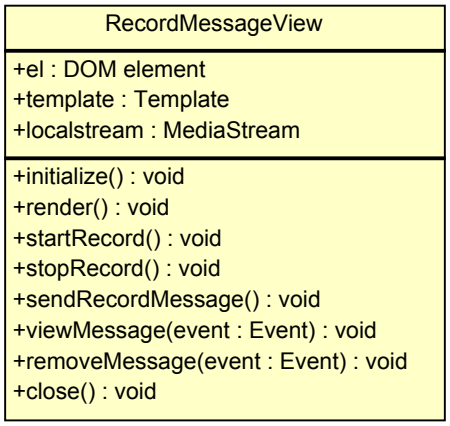

Figura 9: Classe RecordMessageView

Attributi pubblici:

- **[DOM element] el:** elemento DOM al quale è legata la classe
- **[Template] template:** riferimento al template utilizzato
- **[MediaStream] localstream:** stream audio/video locale

Eventi:

- **click button#startRecord:** evento che richiama la funzione *startRecording*
- **click button#stopRecord:** evento che richiama la funzione *stopRecording*
- **click button#sendRecord:** evento che richiama la funzione *sendRecording*
- **click li.message:** evento che richiama la funzione *viewMessage*
- **click button#removeRecord:** evento che richiama la funzione *removeMessage*

Metodi pubblici:

- **[void] initialize():** metodo per inizializzare l'oggetto, si occupa di inizializzare la variabile *localstream* e richiamare il metodo *render*

- **[void] render():** metodo per effettuare la scrittura nella pagina del template RecordMessageTemplate
- **[void] startRecord():** metodo per avviare la registrazione del messaggio, inizializza la variabile *localStream* e inoltre sostituisce il pulsante per avviare la registrazione con il pulsante per fermarla
- **[void] stopRecord():** metodo per fermare la registrazione del messaggio, sostituisce il pulsante per terminare la registrazione con i pulsanti per riavviarla e per inviare il messaggio
- **[void] sendRecordMessage():** metodo per inviare il messaggio, invocando il metodo *sendRecordMessage* di RecordMessageCommunication e ripristinando lo stato iniziale della view
- **[void] viewMessage([Event] event):** metodo per visualizzare il video messaggio selezionato, si occupa anche di aggiornare il template
- **[void] removeMessage([Event] event):** metodo per eliminare il video messaggio selezionato, invoca il metodo *removeRecordMessage* della classe RecordMessageCommunication
- **[void] close():** metodo che chiude l'istanza della classe creata

4.1.9 `mytalk.client.view.SideView`

La classe SideView definisce la struttura, e la conseguente visualizzazione, della pagina web che consente la visualizzazione della lista utenti iscritti al server. La classe è composta da una lista di ContactView

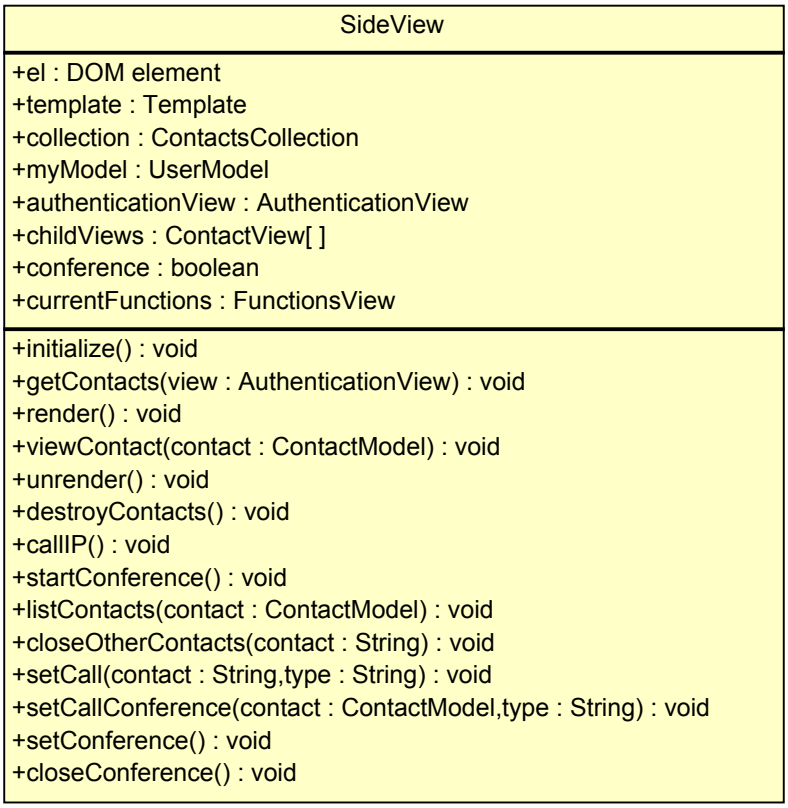

Figura 10: Classe SideView

Attributi pubblici:

- **[DOM element] el:** elemento DOM al quale è legata la classe
- **[Template] template:** riferimento al template utilizzato
- **[ContactsCollection] collection:** riferimento alla classe ContactsCollection
- **[UserModel] myModel:** riferimento alla classe UserModel
- **[AuthenticationView] authenticationView:** riferimento alla classe AuthenticationView
- **[ContactView[]] childViews:** riferimenti alle istanze della classe ContactView

- **[boolean] conference:** variabile booleana che segnala se si è o no in una conferenza
- **[FunctionsView] currentFunctions:** riferimento alla classe FunctionsView

Eventi:

- **click button#callIP:** evento che richiama la funzione *callIP*
- **click button#Conference:** evento che richiama la funzione *startConference*

Metodi pubblici:

- **[void] initialize():** metodo per inizializzare l'oggetto, collega il metodo *render()* della classe alla *collection* in modo da richiamarlo ogni qual volta si aggiunga un elemento a quest'ultima. Predispone le funzioni *acceptCall(Event)* e *viewMessage(Event)* per gestire, rispettivamente, gli eventi *acceptCall* e *viewMessage* inviati da NotificationView. Infine effettua la scrittura, nella pagina, del template SideTemplate con lo stato dell'utente impostato come disconnesso
  - **[void] acceptCall([Event] event):** metodo che invoca il metodo *setCall* avente come parametri l'username del contatto che ha iniziato la chiamata e il tipo di chiamata
  - **[void] acceptCallConference([Event] event):** metodo che invoca il metodo *setCallConference*
- **[void] getContacts([AuthenticationView] view):** metodo che aggiorna la classe dopo l'autenticazione dell'utente, inizializzando *myModel* e *authenticationView* e aggiornando il template con lo stato dell'utente impostato a connesso. Inoltre invoca il metodo *fetchContacts* della classe ContactsCommunication per popolare la lista degli utenti registrati al server
- **[void] render():** metodo per visualizzare l'ultimo utente aggiunto nella collection, per fare ciò invoca il metodo *viewContact*
- **[void] viewContact([ContactModel] contact):** metodo per visualizzare un contatto. Si occupa di creare una ContactView corrispondente al contatto, di aggiungerla a *childViews* e di appenderla nella lista contenente tutti i contatti
- **[void] unrender():** metodo per chiudere la view, per fare ciò interrompe il collegamento tra la *collection* e il metodo *render*, ripristina il template originale e invoca il metodo *destroyContacts*

- **[void] destroyContacts():** metodo per chiudere tutte le ContactView presenti in *childViews* e cancellare i contatti dalla *collection*
- **[void] callIP():** metodo per effettuare una comunicazione attraverso indirizzo IP, per fare ciò inizialmente invoca il metodo *closeOtherContacts*, per controllare che tutte le view presenti nel content siano state chiuse, e inizializza *currentFunctions* passandogli come parametro *From* il valore *IP*
- **[void] startConference():** metodo per effettuare una videoconferenza, per fare ciò inizialmente invoca il metodo *closeOtherContacts*, per controllare che tutte le view presenti nel content siano state chiuse, e inizializza *currentFunctions* passandogli come parametro *From* il valore *Conf.* Inoltre invoca il metodo *listContacts* per visualizzare la lista degli utenti da selezionare per la conferenza
- **[void] listContacts([ContactModel] contact):** metodo per gestire la lista degli utenti da selezionare per una conferenza, se l'utente risulta connesso si crea un ContactView corrispondente all'utente e si appende nella lista degli utenti da selezionare
- **[void] closeOtherContacts([String] contact):** metodo che si occupa di chiudere le istanze di FunctionsView e UserDataView aperte, in modo da garantire che solo una view sia presente nel content. Se il metodo è invocato da un'istanza di ContactView, viene passato come parametro lo username relativo al contatto
- **[void] setCall([String] contact, [String] type):** metodo per impostare la ricezione di una chiamata, i parametri rappresentano lo username del chiamante e il tipo di chiamata. Invoca la funzione *createCall* dell'istanza, della classe ContactView, relativa al contatto
- **[void] setCallConference([ContactModel] contact, [String] type):** metodo che cerca di generare una conferenza con l'utente chiamante. Imposta l'attributo *conference* a *true* ed invoca i metodi *createCallConference([String] type, [ContactModel] contact, [SideView] sideView)* e *metodo currentFunction* della classe stessa
- **[void] setConference():** metodo che imposta l'attributo *conference* a *true*
- **[void] closeConfrence():** metodo che si occupa di impostare a *false* l'attributo *conference* e di chiudere le istanze di FunctionsView e UserDataView aperte tramite l'invocazione del metodo *closeOtherontacts()* della classe stessa

4.1.10 `mytalk.client.view.StatisticsView`

La classe StatisticsView definisce la struttura, e la conseguente visualizzazione, delle pagine web per la visualizzazione di statistiche riguardanti la chiamata

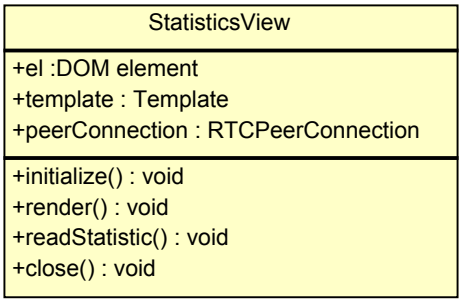

Figura 11: Classe StatisticsView

Attributi pubblici:

- **[DOM element] el:** elemento DOM al quale è legata la classe
- **[Template] template:** riferimento al template utilizzato
- **[RTCPeerConnection] peerConnection:** flusso dati della chiamata

Metodi pubblici:

- **[void] initialize():** metodo per inizializzare l'oggetto, inoltre richiama il metodo *render*
- **[void] render():** metodo per effettuare la scrittura nella pagina del template StatisticsTemplate ed invoca il metodo *readStatistic* per visualizzare le statistiche a video
- **[void] readStatistic():** metodo per la gestione delle statistiche delle chiamate audio/video. Per fare ciò istanzia un listener per gli eventi di tipo *setPeerConn,* collegandolo al metodo *showStatistics*
  - **[void] showStatistics([Event] event):** metodo che si occupa di aggiornare, con la frequenza di un secondo, la durata, i byte di video e audio trasmessi, la latenza e la velocità di trasmissione della chiamata. Riceve i dati necessari dal metodo *getStats* dell'oggetto *peerConnection* e utilizza il metodo *setInterval()* di JavaScript per aggiornarsi
- **[void] close():** metodo che chiude l'istanza della classe creata

4.1.11 `mytalk.client.view.TutorialView`

La classe TutorialView definisce la struttura, e la conseguente visualizzazione, della parte della pagina web che consente la visualizzazione dei video tutorial del prodotto **MyTalk**

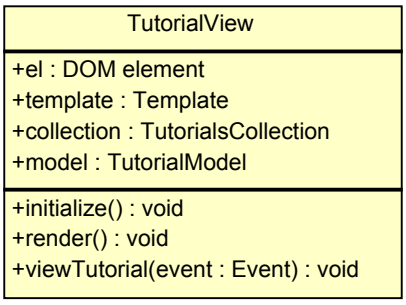

Figura 12: Classe TutorialView

Attributi pubblici:

- **[DOM element] el:** elemento DOM al quale è legata la classe
- **[Template] template:** riferimento al template utilizzato
- **[TutorialsCollection] collection:** lista dei tutorials disponibili
- **[TutorialModel] model:** riferimento al tutorial corrente

Eventi:

- **click a#tutorial:** evento che richiama la funzione *viewTutorial()*
- **click button#prev:** evento che richiama la funzione *viewTutorial()*
- **click buttonn#indice:** evento che richiama la funzione *render()*
- **click button#next:** evento che richiama la funzione *viewTutorial()*

Metodi pubblici:

- **[void] initialize():** metodo per inizializzare l'oggetto, collega il metodo *render* della classe alla *collection* in modo da richiamarlo ogni qual volta si aggiunga un elemento a quest'ultima

- **[void] render():** metodo per effettuare la scrittura nella pagina del template TutorialTemplate, al fine di visualizzare la lista dei tutorials disponibili
- **[void] viewTutorial([Event] event):** metodo per visualizzare il video tutorial selezionato, indicato attraverso un parametro indice di *collection*. Inoltre aggiorna il template in base al video tutorial visualizzato

4.1.12 `mytalk.client.view.UserDataView`

La classe UserDataView definisce la struttura, e la conseguente visualizzazione, della parte di pagina contenente i dati dell'utente autenticato e il collegamento ai metodi per la modifica di tali dati

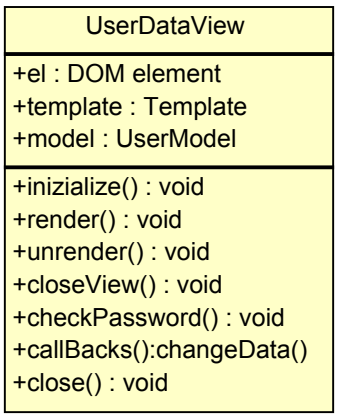

Figura 13: Classe UserDataView

Attributi pubblici:

- **[DOM element] el:** elemento DOM al quale è legata la classe
- **[Template] template:** riferimento al template utilizzato
- **[UserModel] model:** riferimento alla classe UserModel

Eventi:

- **click button#submitChange:** evento che richiama la funzione *checkPassword()*
- **click button#reset:** evento che richiama la funzione *render()*

- **click button#denyChange:** evento che richiama la funzione *unrender()*
- **click button#close:** evento che richiama la funzione *closeView()*

Metodi pubblici:

- **[void] initialize():** metodo per inizializzare l'oggetto, si occupa di richiamare il metodo *render*
- **[void] render():** metodo per effettuare la scrittura nella pagina del template TutorialTemplate
- **[void] unrender():** metodo per rimuovere la vista dei dati dell'utente, invoca il metodo *close*
- **[void] closeView():** metodo che si occupa di rimuovere la vista dei dati dell'utente, invoca il metodo *close*
- **[void] checkPassword():** metodo che provvede a controllare la correttezza della password, per fare ciò invoca il metodo *checkPassword* della classe UserDataCommunication
- **[changeData()] callBacks():** metodo che permette ad altre classi di usufruire del metodo *changeData*
  - **[void] changeData([UserModel] user, [UserDataView] view):** metodo per modificare i dati dell'utente. Per fare ciò, inizialmente, deve verificare che la nuova password, se esistente, coincida con quella di verifica, se supera il controllo invoca il metodo *changeData* di UserDataCommunication, passandogli i dati aggiornati. Altrimenti mostra il messaggio di errore *Le due password inserite non coincidono*
- **[void] close():** metodo che chiude l'istanza della classe creata

4.2 Package Communication

Nel package communication sono presenti le classi che effettuano la comunicazione con il server.

Costituito dalle classi:

- `mytalk.client.communication.AuthenticationCommunication` (vedasi sezione 4.2.1)
- `mytalk.client.communication.CallCommunication` (vedasi sezione 4.2.2)
- `mytalk.client.communication.ChatCommunication` (vedasi sezione 4.2.3)
- `mytalk.client.communication.ContactsCommunication` (vedasi sezione 4.2.4)

- `mytalk.client.communication.FileCommunication` (vedasi sezione 4.2.5)
- `mytalk.client.communication.NotificationCommunication` (vedasi sezione 4.2.6)
- `mytalk.client.communication.RecordMessageCommunication` (vedasi sezione 4.2.7)
- `mytalk.client.communication.TutorialCommunication` (vedasi sezione 4.2.8)
- `mytalk.client.communication.UserDataCommunication` (vedasi sezione 4.2.9)

4.2.1 `mytalk.client.communication.AuthenticationCommunication`

La classe AuthenticationCommunication si occupa di inviare le credenziali inserite al server, e inviare la risposta ricevuta ad AuthenticationView

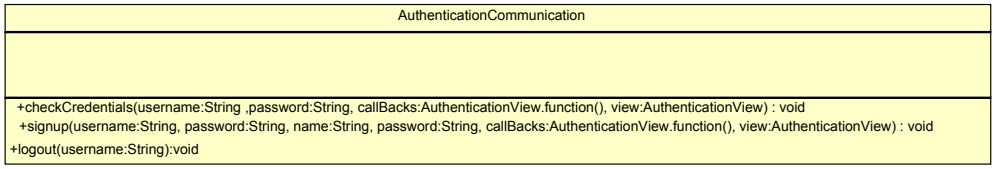

Figura 14: Classe AuthenticationCommunication

Metodi pubblici:

- **[void] checkCredentials([String] username, [String] password, [AuthenticationView.function()] callBacks, [AuthenticationView] view):** metodo che si occupa di controllare la correttezza dello username e della password inserite. Per fare ciò invia al server un pacchetto, contenente username e password e avente come parametro aggiuntivo il tipo *login,* e attende il messaggio di risposta, anch'esso con tipo *login.* Se il messaggio risulta positivo invoca il metodo *callBacks.doLogin* di AuthenticationView
- **[void] signup([String] username, [String] password, [String] name, [String] password, [AuthenticationView.function()] callBacks, [AuthenticationView] view):** metodo che si occupa della registrazione dell'utente. Per fare ciò invia al server un pacchetto contenente tutte le credenziali e avente tipo *signUp*, e attende il messaggio di risposta contrassegnato con il tipo *signUp.* Se il messaggio risulta positivo invoca il metodo *callBacks.doLogin* di AuthenticationView
- **[void] logout([String] username):** metodo che gestisce la disconnessione dell'utente

4.2.2 `mytalk.client.communication.CallCommunication`

La classe CallCommunication si occuperà di avviare la comunicazione tra utenti

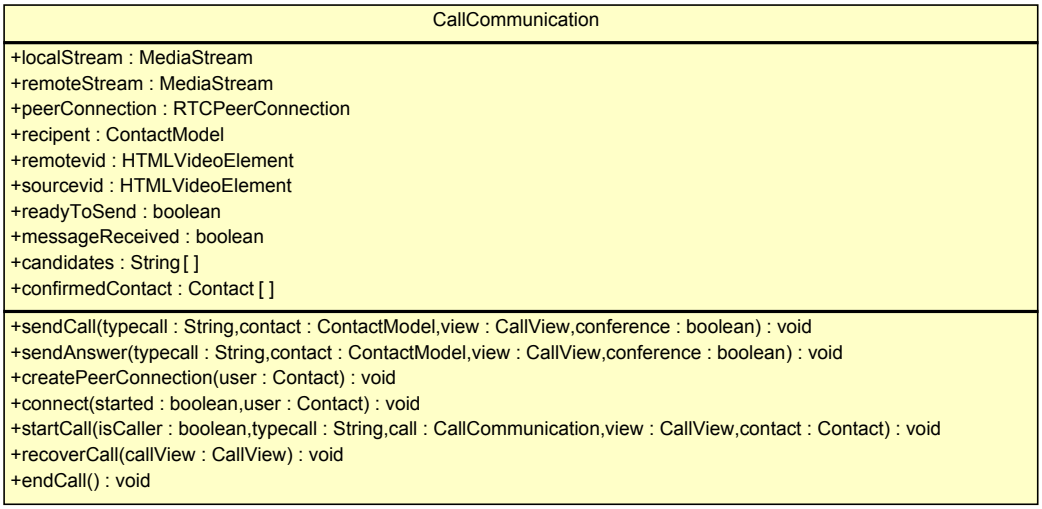

Figura 15: Classe CallCommunication

Attributi pubblici:

- **[MediaStream] localStream:** stream audio/video dell'utente locale
- **[MediaStream] remoteStream:** stream audio/video dell'utente con cui si sta comunicando
- **[RTCPeerConnection] peerConnection:** canale attraverso cui scorre il flusso della chiamata tra due utenti
- **[ContactModel] recipent:** riferimento alla classe ContactModel riguardante l'utente con cui si sta comunicando
- **[HTMLVideoElement] remotevid:** elemento HTMLVideoElement dell'utente con cui si sta comunicando
- **[HTMLVideoElement] sourcevid:** elemento HTMLVideoElement dell'utente locale
- **[boolean] readyToSend:** variabile booleana che indica se è possibile o meno l'invio di *candidates*

- **[boolean] messageReceived:** variabile booleana che indica se l'utente con cui si vuole comunicare è pronto ad accettare *candidates* o meno
- **[String[]] candidates:** array associativo contenente gli ICE candidates, *recipient* e il tipo di messaggio da inviare al server
- **[Contact[]] confirmedContact:** array di contatti contenente tutti i Contact con cui si è in conversazione

Metodi pubblici:

- **[void] sendCall([String] typecall, [ContactModel] contact, [CallView] view, [boolean] conference):** metodo per inizializzare la chiamata, deve avere come parametri il tipo di chiamata, l'istanza della classe ContactModel relativa al contatto, un riferimento a CallView e un booleano che indica se è una chiamata multipla o no. Lancia l'evento *setOnCall,* che NotificationCommunication raccoglie per impostare la variabile *onCalling* a *true*, e invia al server un package di tipo *call* contenente il tipo di chiamata e il contatto che si vuole chiamare. Infine inizializza il listener che si occuperà di attendere la risposta del chiamato e invocare *onAnswer(Event)*
  - **[void] onAnswer([Event] event):** metodo che gestisce la risposta inviata dall'utente che si vuole chiamare, identificato dal tipo *answeredCall*; se la risposta risulta affermativa invoca il metodo *addVideoConference* di CallView, per poi chiamare *startCall*, avente come parametri se l'utente è il chiamante, quindi in questo caso viene inviato true, il tipo di chiamata e i riferimenti alle due classi, altrimenti lancia l'evento *setOnCall,* che NotificationCommunication raccoglie per impostare la variabile *onCalling* a *false*, e comunica all'utente il motivo della non inizializzazione della chiamata
- **[void] sendAnswer([String] typecall, [ContactModel] contact, [CallView] view, [boolean] conference):** metodo per inviare all'utente che ha effettuato la chiamata una risposta positiva e per invocare il metodo *startCall*, avente come parametri se l'utente è il chiamante, quindi in questo caso viene inviato false, il tipo di chiamata, i riferimenti alle due classi e *conference*
  - **[void] onAddConference([Event] event):** metodo che gestisce la risposta da un utente con cui si vuole conversare, che può essere:
    - *addConferenceCaller*: avvia la comunicazione con l'utente inserendolo in *confirmedContact* e invocando i metodi *addVideoConference* e *startCall*
    - *addConferenceAnswer*: avvia la comunicazione con l'utente inserendolo in *confirmedContact* e invocando i metodi *addVideoConference* e *startCall*

    * *endCallEarly*: impedisce l'avvio della chiamata

- **[void] createPeerConnection([Contact] user):** metodo per inizializzare una *peerConnection* con l'utente *user* e aggiungere gli ascoltatori per l'aggiunta e la rimozione dello stream remoto
  - **[void] onRemoteStreamAdded([Event] event):** metodo per impostare *remoteStream* allo stream remoto e mandare l'evento *setPeerConn* che viene raccolto e gestito dalla classe StatisticsView
  - **[void] onRemoteStreamRemoved([Event] event):** metodo per rimuovere *localStream* da *peerConnection* (ancora non supportato da WebRTC)
- **[void] connect([boolean] started, [Contact] user):** metodo per inizializzare una RTCPeerConnection con *user* invocando il metodo *createPeerConnection*, aggiungendo a *peerConnection* il proprio *localStream* e creando un messaggio contente il nostro *sdp* da mandare all'utente con cui si vuole comunicare
- **[void] startCall([boolean] isCaller, [String] typecall, [CallCommunication] call, [CallView] view, [Contact] contact):** metodo per avviare la comunicazione tra due utenti. Imposta *remotevid*, inizializza vari listener e se l'utente è il chiamante imposta il listener *onAnswerSDP*, altrimenti il listener *onOfferSDP*
  - **[void] onOfferSDP([Event] event):** metodo che invoca *createPeerConnection*, aggiunge lo stream locale, chiama *setRemoteDescription* e crea l'*answer* per l'altro utente. Una volta terminati i propri compiti il listener si cancella
  - **[void] onAnswerSDP([Event] event):** metodo che imposta *started* a *true* e invoca *setRemoteDescription*. Una volta terminati i propri compiti il listener si cancella
  - **[void] onCandidate([Event] event):** metodo per impostare i candidati provenienti da un determinato utente all'interno della *peerConnection* associata a quell'utente
  - **[void] onEndCall([Event] event):** metodo per rimuovere gli stream video di altri utenti che abbandonano la chiamata e, in caso *confirmedContact* diventi vuoto, interrompere il *localStream*, segnalare la disponibilità ad altre chiamata e riportare l'utente su FunctionView
  - **[void] onCandidateReady([Event] event):** metodo che riconosce il fatto che l'utente remoto sia pronto a ricevere candidati
- **[void] recoverCall([CallView] callView):** metodo per ripristinare la visione dello stream tramite *addVideoConference* nel caso in cui fosse stata interrotta selezionando un altro utente

- **[void] endCall():** metodo che interrompe *localStream*, chiude *peerConnection*, lancia l'evento *setOnCall,* che NotificationCommunication raccoglie per impostare la variabile *onCalling* a *false,* infine invia al server un pacchetto, contrassegnato dal tipo *endcall* contenente lo username del contatto con cui si era in chiamata e si assicura di chiudere tutte le view e i listener non più necessari

4.2.3 `mytalk.client.communication.ChatCommunication`

La classe ChatCommunication si occupa di comunicare con il server per quanto riguarda il trasferimento e la gestione di messaggi testuali

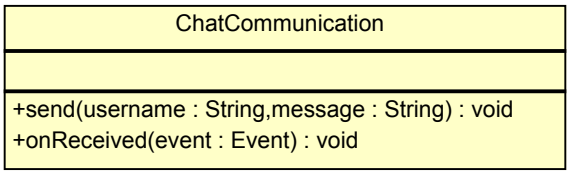

Figura 16: Classe ChatCommunication

Metodi pubblici:

- **[void] send([String] username, [String] message):** metodo che si occupa di inviare un pacchetto al server, contrassegnato con il tipo *sendText* contenente il contatto con cui si sta comunicando via chat e il messaggio che si vuole inviare
- **[void] onReceived([Event] event):** metodo di tipo listener per la ricezione e la gestione di segnali riguardanti i messaggi testuali; se il segnale significa l'arrivo di un nuovo messaggio, è quindi caratterizzati dal tipo *sendText,* lo aggiunge alla collezione di tipo TextMessagesCollection, mentre se indica errori nella consegna del testo, tipo *notDelivered*, avvisa l'utente e imposta lo stato del messaggio, presente nella collection, come *notsent*

4.2.4 `mytalk.client.communication.ContactsCommunication`

La classe ContactsCommunication si occupa di prelevare la lista degli utenti presenti nel server e di aggiornarla nel caso di cambiamenti

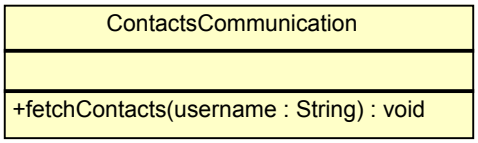

Figura 17: Classe ContactsCommunication

Metodi pubblici:

- **[void] fetchContacts([String] username):** metodo per prelevare dal server la lista dei contatti registrati, per fare ciò invia al server un pacchetto di tipo *getContacts* contenente lo username dell'utente ed avvia il listener onmessage
  - **[void] onmessage([Event] event):** listener che attende la risposta, che consiste in un pacchetto di tipo *getContacts* contenente il numero di contatti e la lista con i loro dati. Per ogni contatto controlla che esso non sia già presente nella lista utenti, se affermativo aggiunge il contatto nella collection, altrimenti aggiorna l'attributo *IP* del *ContactUser* corrispondente

4.2.5 `mytalk.client.communication.FileCommunication`

La classe FileCommunication si occupa di gestire l'invio di file ad un altro utente e la ricezione. Pur essendo pianificata tale classe non è stata implementata

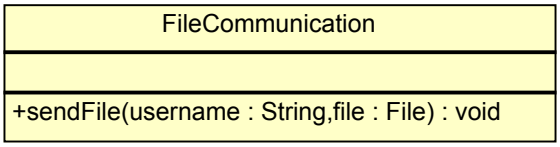

Figura 18: Classe FileCommunication

Metodi pubblici:

- **[void] sendFile([String] username, [File] file):** metodo per inviare un file salvato in locale, invia al server un pacchetto, caratterizzato dal tipo *file,* contenente il file e lo username del destinatario

4.2.6 `mytalk.client.communication.NotificationCommunication`

La classe NotificationCommunication si occupa di comunicare con il server per quanto riguarda le notifiche

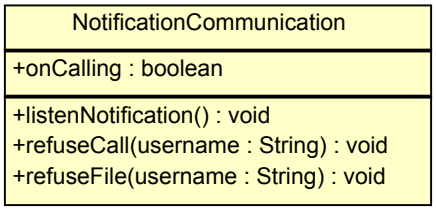

Figura 19: Classe NotificationCommunication

Attributi pubblici:

- **[boolean] onCalling:** variabile che indica se l'utente ha una chiamata in corso

Metodi pubblici:

- **[void] listenNotification():** metodo che si occupa di aggiungere due listener in ascolto di eventi di tipo *setOnCall* e *message*, che richiamano, rispettivamente, i metodi SetOnCall e onNotification presenti nella funzione
  - **[void] SetOnCall([Event] event)** imposta *onCalling* a seconda del tipo racchiuso nel parametro passatogli
  - **[void] onNotification([String] str):** metodo per gestire le notifiche in ingresso, che potranno essere chiamate, messaggi video o file.
    - Se si tratta di una chiamata in arrivo, quindi il pacchetto risulta contrassegnato dal tipo *call*, se l'utente non risulta occupato in altre conversazioni viene creata una istanza NotificationView e impostato il timeout, in modo da terminare la chiamata nel caso non si risponda dopo un determinato intervallo di tempo. Se l'utente risulta invece già in chiamata, si manda un messaggio al server, di tipo *busy* per avvisare il chiamante
    - Se si tratta della terminazione di una chiamata, quindi il pacchetto risulta contrassegnato dal tipo *endcall*, si assegna *false* alla variabile onCalling

  - Se si tratta di un file in arrivo, pacchetto contrassegnato dal tipo *file*, verrà creata e avviata una istanza di NotificationView con i relativi dati
  - Se si tratta di un messaggio video in arrivo, pacchetto contrassegnato dal tipo *record,* viene creato ed aggiunto il messaggio alla collection, successivamente viene creata e avviata una istanza di NotificationView con i relativi dati

- **[void] refuseCall([String] username):** metodo per rispondere ad una richiesta di chiamata con un rifiuto, inviando un pacchetto con tipo *refuseCall*, contenente lo username del chiamante, allo strato transfer
- **[void] refuseFile([String] username):** metodo per rifiutare la ricezione di un file, invia un pacchetto con tipo *refuse*File, contenente lo username del mittente, allo strato transfer

4.2.7 `mytalk.client.communication.RecordMessageCommunication`

La classe RecordMessageCommunication si occupa di registrare, inviare e ricevere i messaggi registrati. Pur essendo pianificata tale classe non è stata implementata

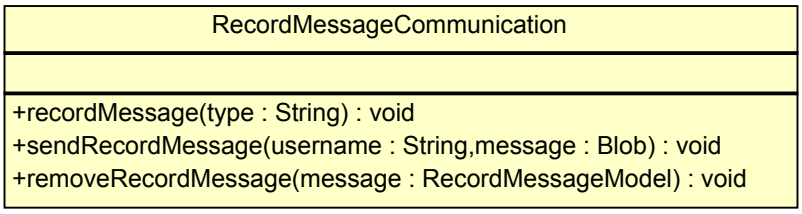

Figura 20: Classe RecordMessageCommunication

Metodi pubblici:

- **[void] recordMessage([String] type)**: metodo per gestire la registrazione del messaggio, type conterrà il tipo di messaggio, e quindi *audio* o *video*
- **[void] sendRecordMessage([String] username, [Blob] message):** metodo per inviare un messaggio video/audio, per fare ciò invia al server un pacchetto con tipo *sendRecord* e contenente il nome del destinatario e il file audio/video

- **[void] removeRecordMessage([RecordMessageModel] message):** metodo per eliminare un messaggio video/audio, invia al server la richiesta, con tipo *removeRecord,* ed elimina il messaggio dalla collection

4.2.8 `mytalk.client.communication.TutorialCommunication`

La classe TutorialCommunication si occupa di comunicare con il server per il prelievo dei tutorial video

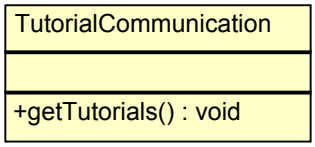

Figura 21: Classe TutorialCommunication

Metodi pubblici:

- **[void] getTutorials():** metodo per prelevare dal server la lista dei tutorial registrati, rappresenta un listener che riceverà un pacchetto, contrassegnato dal tipo *tutorials*, contenente la lista dei tutorial, infine aggiunge ciascuno di questi nella collection

4.2.9 `mytalk.client.communication.UserDataCommunication`

La classe UserDataCommunication si occupa di comunicare con il server per la modifica dei dati dell'utente

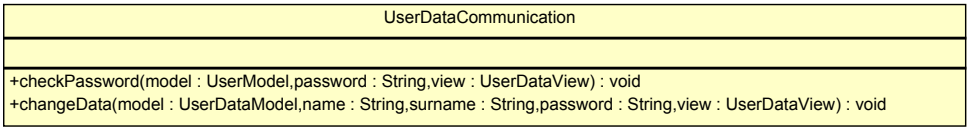

Figura 22: Classe UserDataCommunication

Metodi pubblici:

- **[void] checkPassword([UserModel] model, [String] password, [UserDataView] view):** metodo per controllare che la password di verifica inserita dall'utente corrisponda a quella salvata nella base di dati. Per fare ciò invia al server un pacchetto, contrassegnato dal tipo *checkCredentials* e contenente lo username dell'utente e la password inserita, se il pacchetto di risposta, contrassegnato dallo stesso tipo, è affermativo viene invocato il metodo *view.callBacks.changeData(UserModel, UserDataView)*, altrimenti mostra il messaggio di errore *Password non corretta*
- **[void] changeData([UserModel] model, [String] name, [String] surname, [String] password, [UserDataView] view):** metodo che invia i dati aggiornati al server, racchiudendoli in un pacchetto di tipo *changeData* ed attende la risposta. Se il pacchetto di risposta, contrassegnato dallo stesso tipo, è affermativo vengono aggiornati i dati del model, viene mostrato il messaggio *Operazione riuscita* e si chiama il metodo *render()* di UserDataView, altrimenti viene mostrato un messaggio di errore

4.3 Package Collection

Nel package collection sono presenti le classi che conterranno le liste di oggetti di tipo Model. Non è presente una classe del package Collection per ogni classe del package Model

Costituito dalle classi:

- `mytalk.client.collection.ContactsCollection` (vedasi sezione 4.3.1)
- `mytalk.client.collection.RecordMessagesCollection` (vedasi sezione 4.3.2)
- `mytalk.client.collection.TextMessagesCollection` (vedasi sezione 4.3.3)
- `mytalk.client.collection.TutorialsCollection` (vedasi sezione 4.3.4)

4.3.1 `mytalk.client.collection.ContactsCollection`

La classe ContactsCollection rappresenta la collezione dei contatti registrati

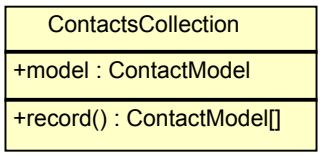

Figura 23: Classe ContactsCollection

Attributi pubblici:

- **[ContactModel] model:** riferimento alla classe ContactModel

Metodi pubblici:

- **[ContactModel[]] record():** metodo per restituire un array contenente tutti i ContactModel della collezione

4.3.2 `mytalk.client.collection.RecordMessagesCollection`

La classe RecordMessagesCollection rappresenta la collezione dei messaggi audio e video ricevuti

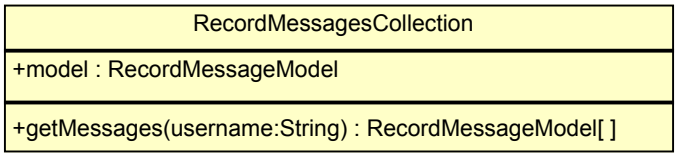

Figura 24: Classe RecordMessagesCollection

Attributi pubblici:

- **[RecordMessageModel] model:** riferimento alla classe RecordMessageModel

Metodi pubblici:

- **[RecordMessageModel[]] getMessages([String] username):** metodo per restituire un array contenente i RecordMessageModel della collezione, avente come mittente lo username passato per parametro

4.3.3 `mytalk.client.collection.TextMessagesCollection`

La classe TextMessagesCollection rappresenta la collezione dei messaggi di testo di una singola conversazione

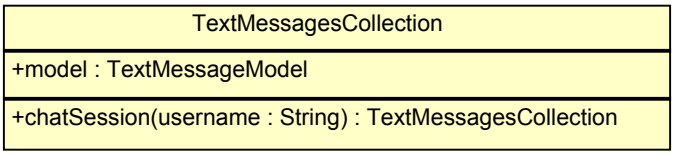

Figura 25: Classe TextMessagesCollection

Attributi pubblici:

- **[TextMessageModel] model:** riferimento alla classe TextMessageModel

Metodi pubblici:

- **[TextMessagesCollection] chatSession([String] username):** metodo per restituire l'intera conversazione con l'utente il cui username è parametro della funzione

4.3.4 `mytalk.client.collection.TutorialsCollection`

La classe TutorialsCollection rappresenta la collezione dei tutorial video disponibili all'utente

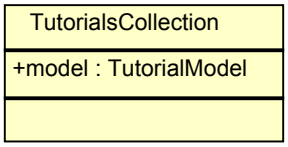

Figura 26: Classe TutorialsCollection

Attributi pubblici:

- **[TutorialModel] model:** riferimento alla classe TutorialModel

4.4 Package Model

Il package model costituisce la parte del sistema dedita alla conservazione dei dati presso il client, per la loro conseguente visualizzazione nell'interfaccia utente.

Costituito dalle classi:

- `mytalk.client.model.ContactModel` (vedasi sezione 4.4.1)
- `mytalk.client.model.RecordMessageModel` (vedasi sezione 4.4.2)
- `mytalk.client.model.TextMessageModel` (vedasi sezione 4.4.3)
- `mytalk.client.model.TutorialModel` (vedasi sezione 4.4.4)
- `mytalk.client.model.UserModel` (vedasi sezione 4.4.5)

4.4.1 `mytalk.client.model.ContactModel`

La classe ContactModel rappresenta un singolo contatto registrato

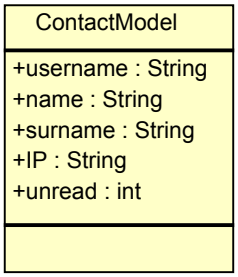

Figura 27: Classe ContactModel

Attributi pubblici:

- **[String] username**: username del contatto
- **[String] name**: nome del contatto
- **[String] surname**: cognome del contatto
- **[String] IP**: indirizzo IP del contatto
- **[int] unread:** numero di messaggi non ancora letti, inviati dal contatto

4.4.2 `mytalk.client.model.RecordMessageModel`

La classe RecordMessageModel rappresenta un messaggio video/audio inviato all'utente

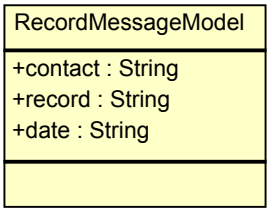

Figura 28: Classe RecordMessageModel

Attributi pubblici:

- **[String] contact:** mittente del messaggio
- **[String] record:** indirizzo url del server in cui è salvato il messaggio
- **[String] date:** data di creazione del messaggio

4.4.3 `mytalk.client.model.TextMessageModel`

La classe TextMessageModel rappresenta un messaggio testuale inviato tra due utenti

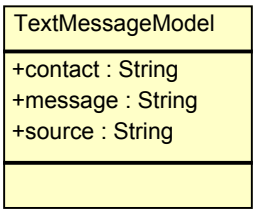

Figura 29: Classe TextMessageModel

Attributi pubblici:

- **[String] contact:** mittente del messaggio testuale
- **[String] message:** contenuto del messaggio testuale
- **[String] source:** stato del messaggio
  - *sent* : se il messaggio è stato inviato al contatto (e *notsent*: se ci sono stati problemi nella consegna)
  - *received:* se il messaggio è stato inviato dal contatto

4.4.4 `mytalk.client.model.TutorialModel`

La classe TutorialModel rappresenta il collegamento ad un video tutorial

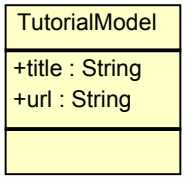

Figura 30: Classe TutorialModel

Attributi pubblici:

- **[String] title:** titolo del tutorial
- **[String] url:** indirizzo url del server in cui è salvato il tutorial

4.4.5 `mytalk.client.model.UserModel`

La classe UserModel rappresenta i dati dell'utente attivo.

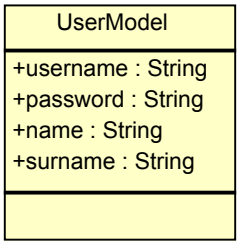

Figura 31: Classe UserModel

Attributi pubblici:

- **[String] username**: username dell'utente
- **[String] password:** password dell'utente
- **[String] name**: nome dell'utente
- **[String] surname**: cognome dell'utente

4.5 template

`mytalk.client.template` costituisce la parte del sistema che definisce ed implementa l'interfaccia web usufruibile dagli utenti mediante pagine web, visualizzata in base alle esigenze. Viene utilizzato il sistema di template fornito da Underscore.js, libreria necessaria per il funzionamento di Backbone.js

4.5.1 `mytalk.client.template.AuthenticationTemplate`

Questo template crea gli elementi dell'interfaccia grafica necessari per effettuare le operazioni di *login* e registrazione e, una volta connessi, per accedere alla sezione di modifica dati ed effettuare il *logout*
Il template è utilizzato da: `mytalk.client.view.AuthenticationView`
Visualizzazione della pagina:

- **Utente non autenticato:**
  - **input#user:** input con tipo text ed id user, in cui l'utente deve inserire il proprio username per effettuare l'accesso
  - **input#password:** input con tipo password ed id password, in cui l'utente deve inserire la propria password per effettuare l'accesso
  - **button#login:** pulsante con id login per effettuare il *login* tramite il metodo connect() della classe AuthenticationView
  - **button#signup:** pulsante con id signup per visualizzare il modulo di registrazione
- **Registrazione:**
  - **input#user:** input con tipo text ed id user, in cui l'utente deve inserire lo username desiderato per effettuare la registrazione
  - **input#password:** input con tipo password ed id password, in cui l'utente deve inserire la password desiderata per effettuare la registrazione

- **input#password2:** input con tipo password ed id password2, in cui l'utente deve inserire nuovamente la propria password per controllare che non ci siano errori in quella inserita precedentemente
- **input#name:** input con tipo text ed id name, in cui l'utente può inserire il proprio nome
- **input#surname:** input con tipo text ed id surname, in cui l'utente può inserire il proprio cognome
- **button#sign:** pulsante con id sign per effettuare la registrazione invocando il metodo signUp() di AuthenticationView
- **button#deny:** pulsante con id deny per annullare la registrazione e chiudere il modulo di registrazione invocando il metodo deny() di AuthenticationView

- **Utente autenticato:**
  - **button#logout:** pulsante con id logout per effettuare il *logout* invocando il metodo disconnect() di AuthenticationView
  - **button#edit:** pulsante con id edit per accedere alla sezione di modifica dei propri dati, ovvero a `mytalk.client.template.UserDataTemplate`

4.5.2 `mytalk.client.template.CallTemplate`

Questo template crea gli elementi dell'interfaccia grafica che l'utente vede durante una chiamata.

- **Il template è utilizzato da:** `myTalk.client.view.CallView`
- **Chiamata avviata:**
  - **video#remotevid:** video con tipologia autoplay ed id remotevid, corrisponde al video dell'utente con cui si sta dialogando
  - **video#sourcevid:** video con tipologia autoplay ed id sourcevid, corrisponde al video locale
  - **button#endCall:** pulsante con id endCall che termina la chiamata invocando il metodo endCall() di CallView

4.5.3 `mytalk.client.template.ChatTemplate`

Questo template crea gli elementi dell'interfaccia grafica che l'utente visualizza quando la chat è attiva.

- **Il template è utilizzato da:** `myTalk.client.view.ChatView`
- **Chat attivata:**
  - **textarea#compose:** area testuale di input con id compose per la scrittura dei messaggi testuali
  - **button#Send:** pulsante con id Send per inviare il messaggio tramite il metodo send() di ChatView

4.5.4 mytalk.client.template.ContactTemplate

Questo template crea gli elementi dell'interfaccia grafica tramite cui l'utente visualizza gli username dei contatti e se tali contatti siano online od offline, e gli elementi per selezionare un contatto da contattare tramite videoconferenza

- **Il template è utilizzato da:** `myTalk.client.view.ContactView`
- **Utente autenticato:**
  - viene verificato l'indirizzo ip per ogni utente registrato al server e, se un utente è connesso viene mostrato "online", altrimenti "offline"
- **Richiesta di video conferenza:**
  - **input#username:** input con tipo checkbox ed id username, con cui l'utente seleziona i partecipanti alla video conferenza

4.5.5 mytalk.client.template.FunctionsTemplate

Questo template crea gli elementi dell'interfaccia grafica necessari per effettuare le operazioni che si possono compiere dopo aver selezionato un utente tramite ip o dalla lista dei contatti.

- **Il template è utilizzato da:** `myTalk.client.view.FunctionsView`
- **Contatto tramite l'inserimento dell'indirizzo IP:**
  - **input#ip:** input con tipo text ed id ip, in cui l'utente deve inserire l'indirizzo ip dell'utente che vuole contattare
  - **button#StartChat:** pulsante con id StartChat che avvia la chat creando una nuova ChatView
  - **button#call:** pulsante con id call che avvia la chiamata tramite il metodo audiocall() di FunctionsView
  - **button#video:** pulsante con id video che avvia la video chiamata tramite il metodo videocall() di FunctionsView
  - **input#record:** input con tipo checkbox e id record per registrare la chiamata invocando il metodo record() di FunctionsView
- **Utente offline:**
  - **button#dataContact:** pulsante con id dataContact che mostra i dettagli del contatto tramite il metodo viewDataContact() di FunctionsView
  - **button#sendVideoText:**pulsante con id sendVideoText che mostra le opzioni per registrare un video messaggio
- **Utente online:**

- **button#call:** pulsante con id call che avvia la chiamata tramite il metodo audiocall() di FunctionsView
- **button#video:** pulsante con id video che avvia la video chiamata tramite il metodo videocall() di FunctionsView
- **input#record:** input con tipo checkbox e id record per registrare la chiamata invocando il metodo record() di FunctionsView

4.5.6 `mytalk.client.template.NotificationTemplate`

Questo template crea gli elementi dell'interfaccia grafica necessari per accettare o rifiutare una chiamata

- **Il template è utilizzato da:** `myTalk.client.view.NotificationView`
- **Notifica della ricezione di una chiamata:**
  - **button#acceptCall:** pulsante con id acceptCall che accetta la chiamata invocando il metodo acceptCall() di NotificationView
  - **button#refuseCall:** pulsante con id refuseCall che rifiuta la chiamata invocando il metodo refuseCall() di NotificationView

4.5.7 `mytalk.client.template.RecordMessageTemplate`

Questo template crea gli elementi dell'interfaccia grafica necessari per registrare un video messaggio

- **Il template è utilizzato da:** `myTalk.client.view.RecordMessageView`
- **Registra un video messaggio:**
  - **video#live_video:** video con tipologia autoplay controls ed id live_video, affinché l'utente possa interrompere la registrazione quando lo desidera

4.5.8 `mytalk.client.template.SideTemplate`

Questo template crea gli elementi dell'interfaccia grafica necessari per la visualizzazione della sidebar

- **Il template è utilizzato da:** `myTalk.client.view.SideView`
- **Sidebar:**
  - **button#callIP:** pulsante con id callIP che permette di contattare una persona tramite l'indirizzo IP invocando il metodo callIP() di SideView
  - **ul#contacts:** lista con id contacts, rappresenta gli utenti registrati presso il server, ciascuno di essi viene mostrato tramite ContactTemplate

- **button#conference:** pulsante con id conference che permette all'utente di selezionare i partecipanti alla conferenza

4.5.9 `mytalk.client.template.StatisticsTemplate`

Questo template crea gli elementi dell'interfaccia grafica riguardanti le statistiche che l'utente visualizza durante una chiamata

- **Il template è utilizzato da:** `myTalk.client.view.StatisticsView`
- **Statistiche:**
  - Durata della chiamata: durata della chiamata mostrata in formato ore:minuti:secondi
  - Byte audio trasmessi: KB trasmessi dall'avvio della chiamata solo audio
  - Byte video trasmessi: KB trasmessi dall'avvio della video chiamata
  - Latenza: tempo di latenza in ms
  - Velocità di trasmissione: KB/s trasmessi

4.5.10 `mytalk.client.template.UserDataTemplate`

Questo template crea gli elementi dell'interfaccia grafica necessari per la modifica dei propri dati salvati nella base di dati

- **Il template è utilizzato da:** `myTalk.client.view.UserDataView`
- **Modifica dati:**
  - **input#name:** input con tipo text ed id name, in cui l'utente può inserire il proprio nome
  - **input#surname:** input con tipo text ed id surname, in cui l'utente può inserire il proprio cognome
  - **input#password:** input con tipo password ed id password, in cui l'utente può inserire la nuova password
  - **input#password2:** input con tipo password ed id password2, in cui l'utente deve inserire nuovamente la nuova password, nel caso in cui l' abbia inserita nel campo precedente
  - **input#oldPassword:** input con tipo password ed id oldPassword, in cui l'utente deve inserire la password attuale per attuare le eventuali modifiche
  - **button#submitChange:** pulsante con id submitChange per confermare le modifiche, invoca il metodo checkPassword() di UserDataView

- **button#reset:** pulsante con id reset che cancella quanto scritto in tutti i campi dati del template invocando il metodo render() di UserDataView
- **button#denyChange:** pulsante con id denyChange che annulla le modifiche e chiude il modulo di modifica dei dati invocando il metodo unrender() di UserDataView

5 Specifica Server

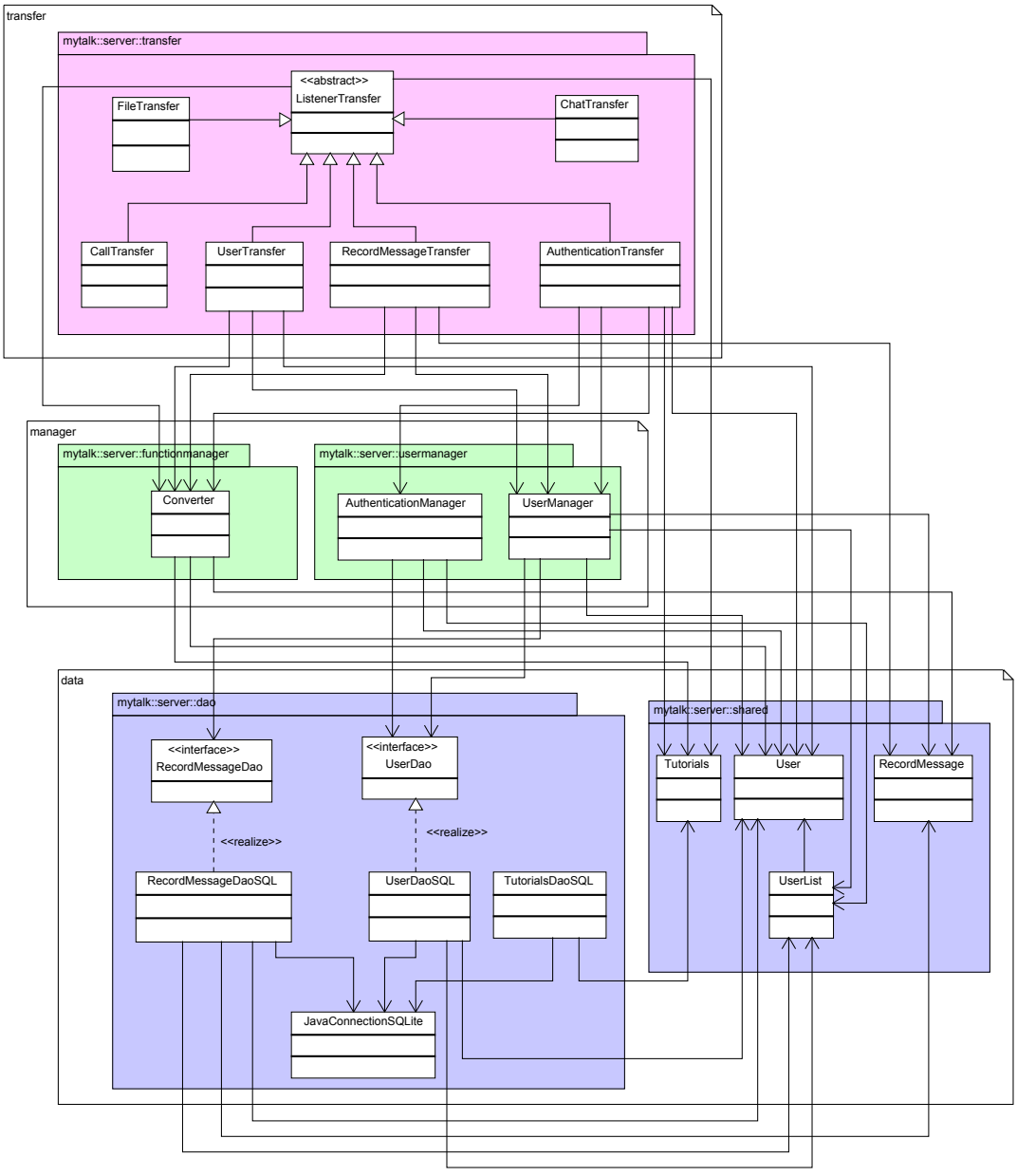

Figura 32: Architettura del server

Il server contiene i seguenti package:

- `mytalk.server.dao` (vedasi sezione 5.1)
- `mytalk.server.shared` (vedasi sezione 5.2)
- `mytalk.server.usermanager` (vedasi sezione 5.3)
- `mytalk.server.functionmanager` (vedasi sezione 5.4)
- `mytalk.server.transfer` (vedasi sezione 5.5)

5.1 Package dao

Il package dao contiene tutte le classi, modellate come Singleton, che si occupano di eseguire le operazioni sul database.

Costituito da:

- `mytalk.server.dao.JavaConnectionSQLite` (vedasi sezione 5.1.1)
- `mytalk.server.dao.RecordMessageDao` (vedasi sezione 5.1.2)
- `mytalk.server.dao.RecordMessageDaoSQL` (vedasi sezione 5.1.3)
- `mytalk.server.dao.TutorialsDaoSQL` (vedasi sezione 5.1.4)
- `mytalk.server.dao.UserDao` (vedasi sezione 5.1.5)
- `mytalk.server.dao.UserDaoSQL` (vedasi sezione 5.1.6)

5.1.1 `mytalk.server.dao.JavaConnectionSQLite`

La classe JavaConnectionSQLite contiene tutti i metodi per l'esecuzione di query all'interno del database.

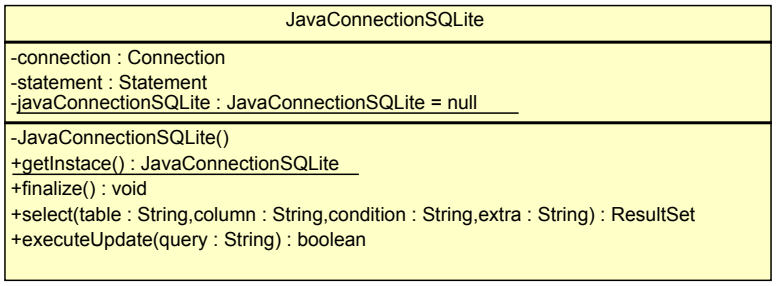

Figura 33: Classe JavaConnectionSQLite

Attributi privati

- **Connection connection:** riferimento alla connessione al database
- **Statement statement:** riferimento alla risorsa impiegata per eseguire query sul database
- **[static] JavaConnectionSQLite javaConnectionSQLite[=null]:** riferimento statico alla classe stessa

Metodi pubblici

- **[static] JavaConnectionSQLite getInstance():** restituisce il riferimento alla classe stessa, serve per garantire che la classe abbia un solo oggetto istanziato
- **void finalize():** funge da distruttore della classe JavaConnectionSQLite. Prima di distruggere una istanza, la connessione al database dev'essere chiusa. Nel caso di fallimento nella chiusura viene lanciata l'eccezione *SQLException* che viene catturata da un catch che stampa a video l'errore riscontrato
- **ResultSet select(String table, String column, String condition, String extra):** ritorna una tupla di valori, della tabella indicata, che rispettano i parametri di controllo indicati. Quindi esegue la query sul database utilizzando i parametri passati:
  - *table:* rappresenta il nome della tabella su cui eseguire la query
  - *column:* rappresenta le colonne che saranno selezionate
  - *condition:* rappresenta la condizione principale applicata per effettuare la selezione
  - *extra:* rappresenta le condizioni extra, se necessarie

  Se la funzione di esecuzione della query non lancia alcuna eccezione, il metodo ritorna un *ResultSet*, altrimenti viene lanciata l'eccezione *SQLException* che viene catturata da un catch e viene ritornato il valore *null*
- **boolean executeUpdate(String query):** si occupa di eseguire la query passata nel database, aggiornandolo. Se la query ha successo il metodo ritorna *true*, altrimenti viene lanciata l'eccezione *SQLException* che viene catturata da un catch e viene ritornato il valore *false*

Metodi privati

- **JavaConnectionSQLite():** costruttore della classe che si occupa di creare la connessione al database. In caso di fallimento della connessione, lancia un eccezione che stampa a video l'errore riscontrato. In particolare, vengono lanciate le eccezioni *ClassNotFoundException* e *SQLException* e vengono tutte catturate da un unico catch che cattura *Exception*

5.1.2 `mytalk.server.dao.RecordMessageDao`

L'interfaccia RecordMessageDao gestisce tutti i dati che riguardano i messaggi differiti inviati ai vari utenti. Fornisce l'interfaccia minima necessaria a tutte le classi derivate che dovranno offrire tale servizio.

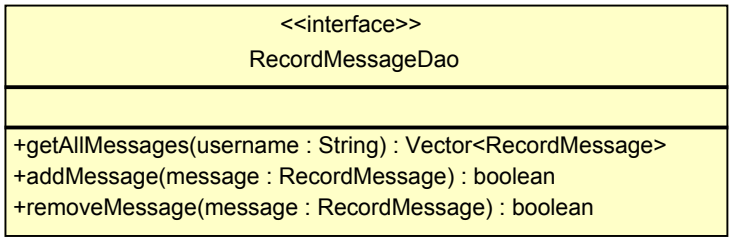

Figura 34: Interfaccia RecordMessageDao

Metodi pubblici

- **Vector<RecordMessage> getAllMessages(String username):** ritorna il vettore dei messaggi differiti, presenti nel database, che sono stati mandati all'utente con username corrispondente a quello passato per parametro
- **boolean addMessage(RecordMessage message):** aggiunge un messaggio differito nel database, se l'operazione ha buon termine restituisce *true*, altrimenti *false*
- **boolean removeMessage(RecordMessage message):** elimina un messaggio differito dal database, se l'operazione ha buon termine restituisce *true*, altrimenti *false*

5.1.3 `mytalk.server.dao.RecordMessageDaoSQL`

La classe RecordMessageDaoSQL si occupa di gestire tutti i dati riguardanti i messaggi differiti.

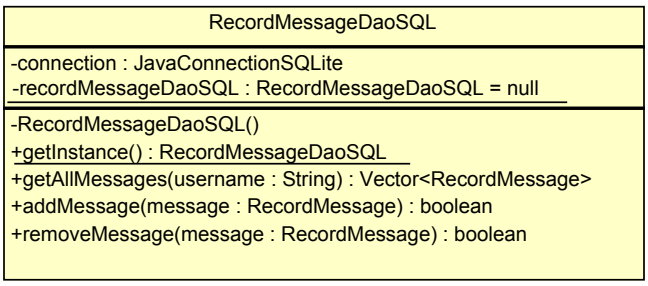

Figura 35: Classe RecordMessageDaoSQL

Attributi privati

- **JavaConnectionSQLite connection:** riferimento all'istanza della classe JavaConnectionSQLite che contiene i riferimenti al server
- **[static] RecordMessageDaoSQL recordMessageDaoSQL[=null]:** riferimento statico alla classe stessa

Metodi pubblici

- **[static] RecordMessageDaoSQL getInstance():** restituisce il riferimento alla classe stessa, serve per garantire che la classe abbia un solo oggetto istanziato
- **Vector<RecordMessage> getAllMessages(String username):** invoca il metodo *select* della classe JavaConnectionSQLite, passando come parametri *(RecordMessageDataSQL,*, addressee=username,)*, per trovare tutti i messaggi che come destinatario avranno l'utente con lo username corrispondente a quello passato per parametro. Se la query restituisce almeno un messaggio differito, lo inserisce nel vettore dell'utente. Se viene lanciata una eccezione di tipo *SQLException* viene catturata dal metodo stesso e vengono restituiti tutti i messaggi aggiunti fino a quel momento
- **boolean addMessage(RecordMessage message):** aggiunge un messaggio differito nel database, invocando il metodo *executeUpdate* della classe JavaConnectionSQLite, passando come parametro la query per l'inserimento del messaggio nella tabella *RecordMessageDataSQL.* Infine restituirà un booleano che indicherà il successo o il fallimento dell'operazione
- **boolean removeMessage(RecordMessage message):** metodo che elimina un messaggio differito dal database, invocando il metodo *executeUpdate* della classe JavaConnectionSQLite, passando come parametro la query per l'eliminazione del messaggio nella tabella *RecordMessageDataSQL.* Infine restituirà un booleano che indicherà il successo o il fallimento dell'operazione

Metodi privati

- **RecordMessageDaoSQL():** costruttore della classe che inizializza *connection*, invocando *JavaConnectionSQLite.getInstance()*

5.1.4 `mytalk.server.dao.TutorialsDaoSQL`

La classe TutorialsDaoSQL si occupa di gestire i tutorial presenti nel database.

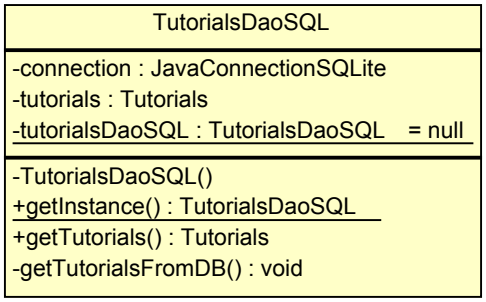

Figura 36: Classe TutorialsDaoSQL

Attributi privati

- **JavaConnectionSQLite connection:** riferimento all'istanza della classe JavaConnectionSQLite contenente i riferimenti al server
- **Tutorials tutorials:** riferimento alla lista dei tutorials presenti nel server
- **[static] TutorialsDaoSQL tutorialsDaoSQL[=null]:** riferimento statico alla classe stessa

Metodi pubblici

- **[static] TutorialsDaoSQL getInstance():** restituisce il riferimento alla classe stessa, serve per garantire che la classe abbia un solo oggetto istanziato
- **Tutorials getTutorials():** restituisce la lista di tutorials presenti in *tutorials*[1]

[1] I tutorial non vengono presi direttamente dal database, perché nel caso di più utenti che accedono contemporaneamente si creerebbero più copie degli stessi dati, sprecando spazio. In questo modo invece lo spazio occupato dai tutorial nel server resterà fisso, qualsiasi sia il numero degli utenti che accedono.

Metodi privati

- **TutorialsDaoSQL():** costruttore della classe che inizializza *connection*, invocando *JavaConnectionSQLite.getInstance()*, inoltre utilizza il metodo *select* della classe JavaConnectionSQLite, passando come parametri *(TutorialDataSQL, count(*) as total, ,)*, per individuare il numero di tutorial presenti nel database e inizializza *tutorials* passando come parametro al costruttore il numero così ottenuto. Se viene lanciata un eccezione di tipo *SQLException* si utilizza 10 come default per il costruttore. Infine invoca il metodo *getTutorialsFromDB()*
- **void getTutorialsFromDB():** utilizza il metodo *select* della classe JavaConnectionSQLite, passando come parametri *(TutorialDataSQL, *, ,)*, per ottenere tutti i tutorial presenti nel database, e li inserisce in *tutorials* utilizzando il metodo *insert* della classe Tutorials. Se viene lanciata un'eccezione di tipo *SQLException* viene catturata ma non viene gestita, semplicemente non si prosegue nell'aggiunta

5.1.5 `mytalk.server.dao.UserDao`

L'interfaccia UserDao gestisce tutti i dati che riguardano gli utenti. Fornisce l'interfaccia minima necessaria a tutte le classi derivate che dovranno offrire tale servizio.

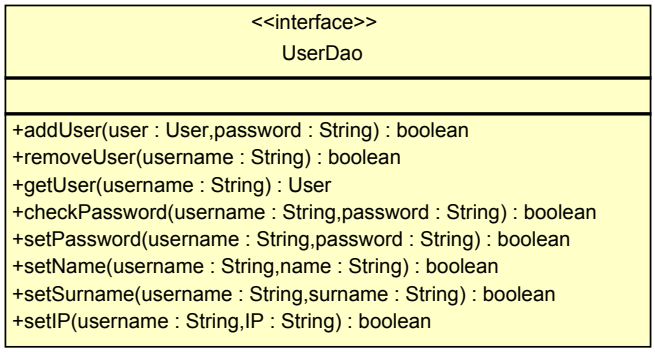

Figura 37: Interfaccia UserDao

Metodi pubblici

- **boolean addUser(User user, String password):** aggiunge un utente nel database
- **boolean removeUser(String username):** elimina un utente dal database

- **User getUser(String username):** restituisce l'utente avente username corrispondente a quello dato
- **boolean checkPassword(String username, String password):** controlla la correttezza dei parametri dati
- **boolean setPassword(String username, String password):** imposta la password dell'utente, avente lo username corrispondente a quello passato per parametro, con il nuovo valore. La password viene salvata in chiaro.
- **boolean setName(String username, String name):** imposta il nome dell'utente, avente lo username corrispondente a quello passato per parametro, con il nuovo valore
- **boolean setSurname(String username, String surname):** imposta il cognome dell'utente, avente lo username corrispondente a quello passato per parametro, con il nuovo valore
- **boolean setIP(String username, String IP):** imposta l'indirizzo IP dell'utente, avente lo username corrispondente a quello passato per parametro, con il nuovo valore

5.1.6 `mytalk.server.dao.UserDaoSQL`

La classe UserDaoSQL si occupa di gestire tutti i dati che riguardano gli utenti.

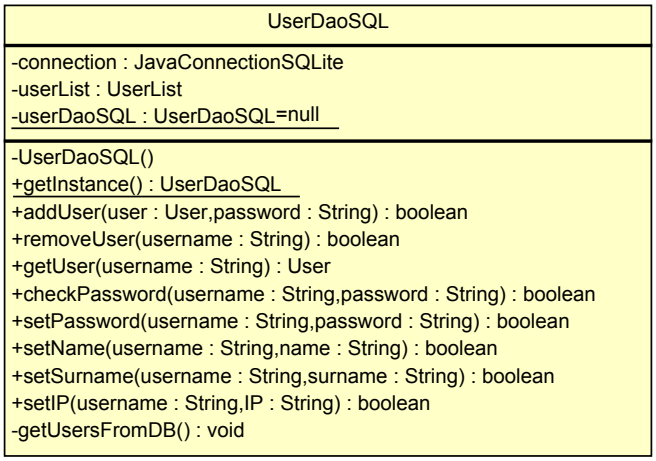

Figura 38: Classe UserDaoSQL

Attributi privati

- **JavaConnectionSQLite connection:** riferimento all'istanza della classe JavaConnectionSQLite contenente i riferimenti al server
- **UserList userList:** riferimento alla lista degli utenti registrati nel database
- **[static] UserDaoSQL userDaoSQL[=null]**: riferimento statico alla classe stessa

Metodi pubblici

- **[static] UserDaoSQL getInstance():** restituisce il riferimento alla classe stessa, serve per garantire che la classe abbia un solo oggetto istanziato
- **boolean addUser(User user, String password):** aggiunge un utente nel database, invocando il metodo *executeUpdate* della classe JavaConnectionSQLite, passando come parametro la query per l'inserimento dei dati nella tabella *UserDataSQL*, in cui la password viene memorizzata in chiaro. Infine restituisce un booleano che indicherà il successo o il fallimento dell'operazione
- **boolean removeUser(String username):** rimuove dal database l'utente con username corrispondente a quello passato per parametro, per fare ciò invoca il metodo *executeUpdate* della classe JavaConnectionSQLite, passando come parametro la query per l'eliminazione dell'utente dalla tabella *UserDataSQL.* Infine restituisce un booleano che indicherà il successo o il fallimento dell'operazione
- **User getUser(String username):** invoca il metodo *select* della classe JavaConnectionSQLite, passando come parametri *(UserDataSQL,*, username=username,),* per trovare le informazioni dell'utente con lo username corrispondente a quello passato per parametro. Se la query ha successo crea un'oggetto user, avente come dati quelli ottenuti, e lo restituisce. Se, invece, viene lanciata una eccezione di tipo *SQLException* viene catturata dal metodo stesso e viene restituito *null*
- **boolean checkPassword(String username, String password):** controlla nel database la corrispondenza dello username passato con la password passata, per fare ciò utilizza il metodo *select* della classe JavaConnectionSQLite, passando come parametri *(UserDataSQL,*, username=username AND (password=password),).* Le password sono salvate in chiaro. Restituisce un booleano che rappresenta il successo o il fallimento dell'operazione
- **boolean setPassword(String username, String password):** modifica nel database la password dell'utente, avente lo username corrispondente a quello passato per parametro, invocando il metodo *executeUpdate* della

classe JavaConnectionSQLite, tramite il quale la password viene salvata in chiaro. Restituisce un booleano che rappresenta il successo o il fallimento dell'operazione

- **boolean setName(String username, String name):** modifica nel database il nome dell'utente, avente lo username corrispondente a quello passato per parametro, invocando il metodo *executeUpdate* della classe JavaConnectionSQLite. Restituisce un booleano che rappresenta il successo o il fallimento dell'operazione
- **boolean setSurname(String username, String surname):** modifica nel database il cognome dell'utente, avente lo username corrispondente a quello passato per parametro, invocando il metodo *executeUpdate* della classe JavaConnectionSQLite. Restituisce un booleano che rappresenta il successo o il fallimento dell'operazione
- **boolean setIP(String username, String IP):** modifica nel database l'indirizzo IP dell'utente, avente lo username corrispondente a quello passato per parametro, invocando il metodo *executeUpdate* della classe JavaConnectionSQLite. Restituisce un booleano che rappresenta il successo o il fallimento dell'operazione

Metodi privati

- **UserDaoSQL():** costruttore della classe che inizializza *connection*, invocando *JavaConnectionSQLite.getInstance(),* e *userList*, invocando il metodo *UserList.getInstance()*, infine invoca il metodo *getUsersFromDB()*
- **void getUsersFromDB():** utilizza il metodo *select* della classe JavaConnectionSQLite, passando come parametri *(UserDataSQL, *, ,)*, per ottenere tutti gli utenti presenti nel database, e li inserisce in *userList* utilizzando il metodo *addUser* della classe UserList. Se viene lanciata un'eccezione di tipo *SQLException* viene catturata ma non viene gestita, semplicemente non si prosegue nell'aggiunta

5.2 Package shared

Il package shared contiene le classi contenenti le informazioni condivise fra i vari strati del server.

Costituito dalle classi:

- `mytalk.server.shared.RecordMessage` (vedasi sezione 5.2.1)
- `mytalk.server.shared.User` (vedasi sezione 5.2.2)
- `mytalk.server.shared.UserList` (vedasi sezione 5.2.3)
- `mytalk.server.shared.Tutorials` (vedasi sezione 5.2.4)

5.2.1 `mytalk.server.shared.RecordMessage`

La classe RecordMessage rappresenta i dati riguardanti un messaggio differito.

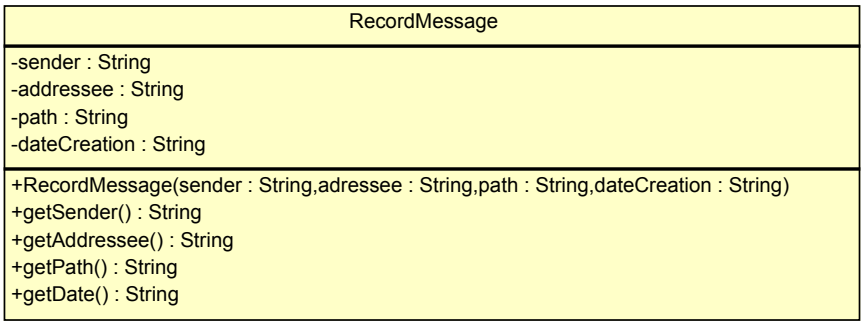

Figura 39: Classe RecordMessage

Attributi privati:

- **String sender:** rappresenta il mittente, colui che crea il messaggio
- **String addressee:** rappresenta il destinatario, colui che riceve il messaggio
- **String path:** rappresenta l'indirizzo dove si trova il messaggio differito all'interno del server
- **String dateCreation:** rappresenta la data di creazione del messaggio

Metodi pubblici:

- **RecordMessage(String sender, String addressee, String path, String dateCreation):** costruttore che imposta i valori degli attributi secondo i valori dei parametri passati
- **String getSender():** ritorna il mittente del messaggio
- **String getAddressee():** ritorna il destinatario del messaggio
- **String getPath():** ritorna l'indirizzo dove si trova il messaggio differito all'interno del server
- **String getDate():** ritorna la data di creazione del messaggio

5.2.2 `mytalk.server.shared.User`

La classe User rappresenta un utente del server

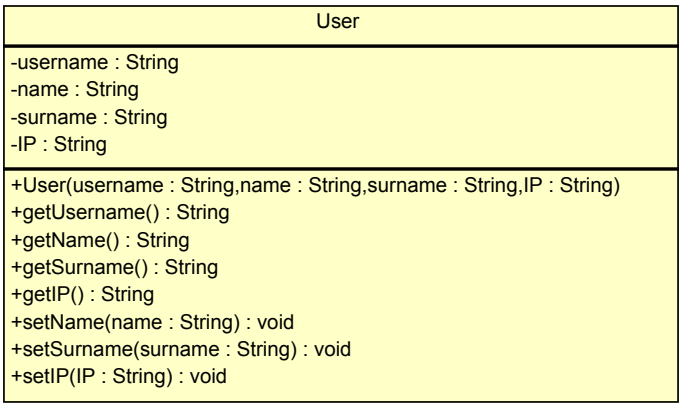

Figura 40: Classe User

Attributi privati

- **String username:** rappresenta lo username dell'utente
- **String name:** rappresenta il nome dell'utente
- **String surname:** rappresenta il cognome dell'utente
- **String IP:** rappresenta l'indirizzo IP dell'utente[2]

Metodi pubblici

- **User(String username, String name, String surname, String IP):** costruttore che imposta i valori degli attributi secondo il valore dei parametri passati
- **String getUsername():** ritorna lo username dell'utente
- **String getName():** ritorna il nome dell'utente
- **String getSurname():** ritorna il cognome dell'utente
- **String getIP():** ritorna l'indirizzo IP dell'utente
- **void setName(String name):** imposta il nome dell'utente con il nuovo valore

[2]Se la stringa è uguale a 0 significa che l'utente in questione non è in linea, se è diverso l'utente è in linea.

- **void setSurname(String surname):** imposta il cognome dell'utente con il nuovo valore
- **void setIP(String IP):** imposta l'indirizzo IP dell'utente con il nuovo valore

5.2.3 `mytalk.server.shared.UserList`

La classe UserList rappresenta la lista degli utenti registrati al server, poiché deve esistere un'unica istanza contenente tutti gli utenti, questa classe è strutturata come Singleton.

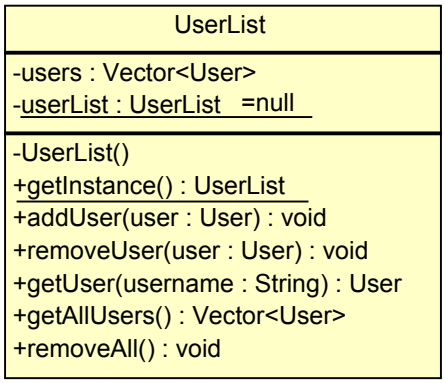

Figura 41: Classe UserList

Attributi privati

- **Vector<User> users:** rappresenta la lista degli utenti
- **[static] UserList userList[=null]:** riferimento statico alla classe stessa

Metodi pubblici

- **[static] UserList getInstance():** restituisce il riferimento alla classe stessa, serve per garantire che la classe abbia un solo oggetto istanziato
- **void addUser(User user):** inserisce un nuovo utente nella lista degli utenti
- **void removeUser(User user):** elimina un utente dal vettore di utenti
- **User getUser(String username):** restituisce l'utente corrispondente ad un dato username, se tale utente non è presente nella lista restituisce *null*

- **Vector<User> getAllUsers():** metodo che restituisce tutta la lista di utenti
- **void removeAll():** metodo che rimuove tutti gli utenti dal vettore

Metodi privati

- **UserList():** costruttore della classe che inizializza il vettore *users*

5.2.4 `mytalk.server.shared.Tutorials`

La classe Tutorials rappresenta la lista dei tutorials presenti nel server.

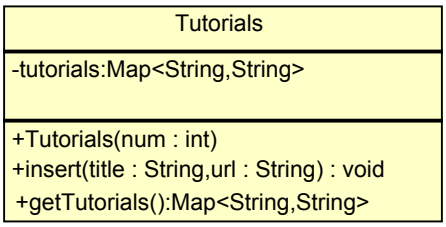

Figura 42: Classe Tutorials

Attributi privati

- **Map<String, String> tutorials:** rappresenta la lista dei tutorials, come chiave di identificazione viene usato il titolo del tutorial

Metodi pubblici

- **Tutorials(int num):** costruttore che inizializza la lista di tutorials della dimensione passatagli dal parametro
- **void insert(String title, String url):** inserisce un nuovo tutorial nella lista
- **Map<String, String> getTutorials():** restituisce la lista dei tutorials

5.3 Package usermanager

Il package usermanager del server si occupa delle funzionalità di gestione e trasferimento di tutti i dati riguardanti gli utenti

Costituito dalle classi:

- `mytalk.server.usermanager.AuthenticationManager` (vedasi sezione 5.3.1)
- `mytalk.server.usermanager.UserManager` (vedasi sezione 5.3.2)

5.3.1 mytalk.server.usermanager.AuthenticationManager

La classe AuthenticationManager si occupa di passare allo strato dao i dati necessari per la verifica dell'autenticazione, e di effettuare i controlli necessari.

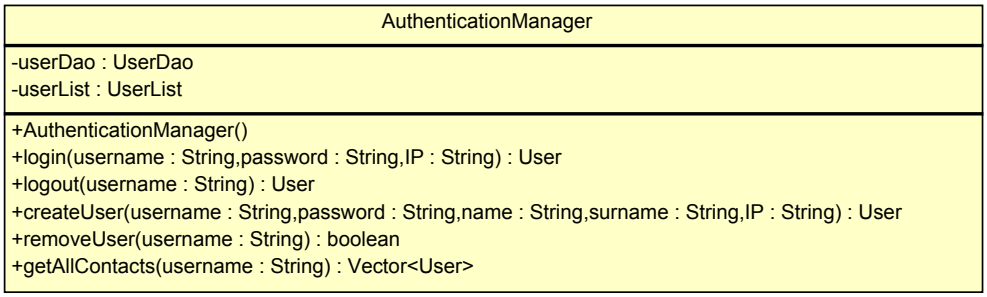

Figura 43: Classe AuthenticationManager

Attributi privati

- **UserDao userDao:** riferimento alla istanza della classe UserDaoSQL contenente i dati per effettuare l'autenticazione
- **UserList userList:** riferimento alla istanza della classe UserList contenente la lista di utenti salvati nel server

Metodi pubblici

- **AuthenticationManager():** costruttore della classe, inizializza gli attributi della classe richiamando i metodi *getInstance* di ciascuna delle due classi UserDaoSQL e UserList
- **User login(String username, String password, String IP):** inizialmente invoca il metodo *getUser* della classe UserDaoSQL per controllare l'esistenza dell'utente nel database. Se non esiste viene lanciata un'eccezione, di tipo generico *Exception*, avente messaggio *Username errato*. Altrimenti si prosegue controllando la validità della password, salvata in chiaro, utilizzando il metodo *checkPassword* della classe UserDaoSQL, se la password è errata si lancerà un'altra eccezione, di tipo generico *Exception*, avente come messaggio *Password errata*. Altrimenti si modificherà l'indirizzo IP dell'utente con il metodo *setIP* della classe UserDao e si controllerà se l'utente è presente anche nella lista del server, utilizzando il metodo *getUser* della classe UserList. Se l'utente non è presente si aggiunge l'utente ottenuto precedentemente con il metodo *getUser* di UserDaoSQL, utilizzando il metodo *addUser* di UserList. Infine si modificherà anche l'IP della coppia presente nella classe UserList, utilizzando il metodo *setIP* della classe

- **User logout(String username):** controlla l'esistenza dello username nella base di dati, utilizzando il metodo *getUser* della classe UserDaoSQL. Se l'utente è presente invoca i metodi *setIP* delle classi UserDaoSQL e UserList per modificare l'indirizzo IP ad una stringa contenente *0*, che rappresenta un utente non connesso. Infine restituisce l'utente ottenuto dalla prima operazione

- **User createUser(String username, String password, String name, String surname, String IP):** inizialmente controlla se l'username passato per parametro corrisponde già a un utente presente nel database, se la risposta è affermativa lancia un eccezione, di tipo *Exception,* avente come messaggio *Username utilizzato da un altro utente.* Altrimenti crea un oggetto di tipo User, utilizzando i parametri passati, ad eccezion fatta per password *e* invoca il metodo *addUser* della classe UserDaoSQL per aggiungere l'utente alla base di dati, se l'operazione fallisce si lancia un'eccezione, di tipo *Exception*, avente come messaggio *Operazione di registrazione fallita.* Altrimenti aggiunge il nuovo utente in UserList e lo restituisce

- **boolean removeUser(String username):** elimina un utente registrato dal database, utilizzando i metodi *removeUser* delle classi UserDaoSQL e UserList. All'inizio effettua un controllo se l'username è registrato nel database, se così non fosse restituisce semplicemente *true,* altrimenti procede con l'operazione di eliminazione

- **Vector<User> getAllContacts(String username):** restituisce tutti gli utenti presenti nel server, utilizzando il metodo *getAllUsers* della classe UserList, ad eccezione di quello avente username corrispondente a quello passato per parametro[3]

5.3.2 `mytalk.server.usermanager.UserManager`

La classe UserManager si occupa di gestire la modifica dei dati degli utenti e dei messaggi differiti.

[3]Gli utenti non vengono presi direttamente dal database, perché nel caso di più utenti che accedono contemporaneamente si creerebbero più copie degli stessi dati, sprecando spazio. In questo modo invece lo spazio occupato dagli utenti nel server resterà fisso, e ciò equivarrà allo spazio occupato dall'istanza di UserList, qualsiasi sia il numero degli utenti che accedono.

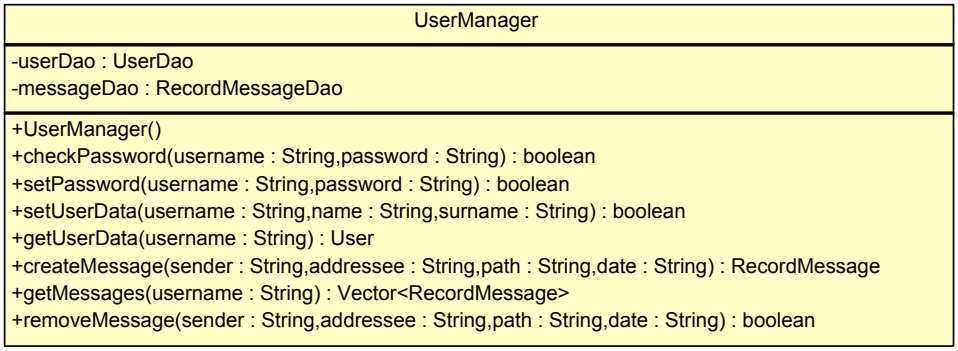

Figura 44: Classe UserManager

Attributi privati

- **UserDao userDao:** riferimento alla istanza della classe UserDaoSQL, per eseguire le operazioni sull'utente e per recuperare la lista di tutti gli utenti registrati nel server
- **RecordMessageDao recordMessageDao:** riferimento alla istanza della classe RecordMessageDaoSQL per permettere le operazioni sui messaggi registrati

Metodi pubblici

- **UserManager():** costruttore della classe, inizializza gli attributi della classe richiamando i metodi *getInstance* di ciascuna delle due classi UserDaoSQL e RecordMessageDaoSQL
- **boolean checkPassword(String username, String password):** controlla la corrispondenza dello username passato con la password passata in chiaro, invocando il metodo *checkPassword* della classe UserDaoSQL
- **boolean setPassword(String username, String password):** inizialmente controlla la presenza dello username nel database, invocando il metodo *getUser* della classe UserDaoSQL. Se l'utente non è presente, e quindi il metodo ha restituito *null*, lancia un'eccezione, di tipo generico *Exception*, avente messaggio *Username errato*. Altrimenti modifica la password esistente dell'utente trovato, inserendo la password passata per parametro in chiaro. Per fare ciò utilizza il metodo *setPassword* della classe UserDao
- **boolean setUserData(String username, String name, String surname):** inizialmente controlla la presenza dello username nel database, invocando il metodo *getUser* della classe Username. Se l'utente non è presente, e quindi il metodo ha restituito *null*, lancia un'eccezione, di tipo

generico *Exception*, avente messaggio *Username errato*. Altrimenti controlla se effettivamente il nome e il cognome, passati per parametro, sono diversi da quelli presenti nel database, se ciò risulta falso restituisce *true,* altrimenti li modifica nel database invocando i metodi *setName* e *setSurname* della classe UserDao. Infine cerca l'utente in UserList, creando un istanza locale della classe, e inizializzandola con il metodo *getInstance*, se è presente modifica i dati, altrimenti aggiunge l'utente nella lista con i dati già modificati

- **User getUserData(String username):** ritorna l'utente corrispondente al dato username, presente nel database, invocando il metodo *getUser* della classe UserDao
- **RecordMessage createMessage(String sender, String addressee, String path, String date):** inizialmente controlla che il destinatario esista, e in caso contrario lancia un'eccezione di tipo *Exception*, avente come messaggio *Destinatario inesistente*. In seguito crea un oggetto di tipo RecordMessage con i dati passati per parametro e aggiunge il messaggio nella base di dati, invocando il metodo *addMessage* della classe RecordMessageDao. Se l'operazione fallisce lancia un'eccezione di tipo *Exception*, avente come messaggio *Errore nella registrazione del messaggio*. Se l'operazione di inserimento ha buon esito, controlla se il destinatario del messaggio è in linea, se la risposta è affermativa restituisce il messaggio altrimenti restituisce *null*
- **Vector<RecordMessage> getMessages(String username):** ritorna il vettore dei messaggi differiti che sono stati mandati all'utente avente username corrispondente a quello dato. Per fare ciò invoca il metodo *getAllMessages* della classe RecordMessageDaoSQL
- **boolean removeMessage(String sender, String addressee, String path, String date):** rimuove un messaggio registrato avente come attributi quelli passati per parametro, per fare ciò crea inizialmente un oggetto di tipo RecordMessage, con i parametri passati, e invoca il metodo *removeMessage* della classe RecordMessageDaoSQL

5.4 Package functionmanager

Il package functionmanager contiene tutte le classi che si occupano di convertire le strutture dati più complesse in stringhe JSON per lo strato transfer.

Costituito dalla classe:

- `mytalk.server.functionmanager.Converter` (vedasi sezione 5.4.1)

5.4.1 `mytalk.server.functionmanager.Converter`

La classe Converter restituisce in forma di stringa JSON il vettore di utenti passato per argomento

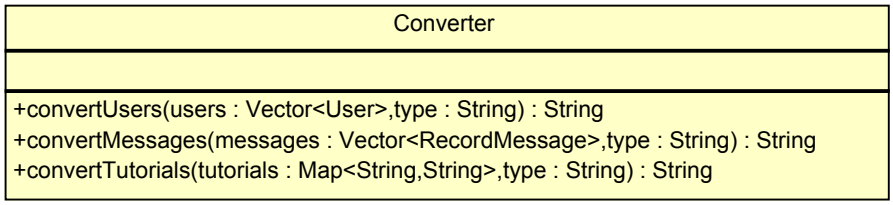

Figura 45: Classe Converter

Metodi pubblici

- **String convertUsers(Vector<User> users, String type):** metodo che converte in forma di stringa JSON il vettore di utenti passato per argomento, inserendo inizialmente la stringa type e il numero di utenti presenti nel vettore passato
  - Esempio 1:
    - users.size() == 1
      - user ha come dati: username = "ClockWork", name = "team", surname vuoto, IP = "0"
    - type = "getContacts"
    - la stringa risultante è quindi {type: "getContacts", size: "1", username0: "ClockWork", name0: "team", surname0: "", IP0: "0"}
  - Esempio 2:
    - users.size() == 2
      - il primo utente è quello indicato nell'esempio 1
      - secondo utente ha come dati : username = "ClockworkTeam", name vuoto, surname vuoto, IP = "0"
    - type = "getContacts"
    - la stringa risultante è quindi {type: "getContacts", size: "2", username0: "ClockWork", name0: "team", surname0: "", IP0: "0", username1: "ClockworkTeam", name1: "", surname1: "", IP1: "0"}
- **String convertMessages(Vector<RecordMessage> messages, String type):** metodo che converte in forma di stringa JSON il vettore di messaggi passato per argomento, inserendo inizialmente la stringa type e il

numero di messaggi presenti nel vettore. La stringa risultante è analoga a quella mostrata negli esempi 1 e 2, per il metodo *convertUsers*, con la differenza che al posto degli attributi degli utenti, si hanno quelli dei messaggi

- **String convertTutorials(Map<String, String> tutorials, String type):** metodo che converte in forma di stringa JSON i tutorials passati per argomento, inserendo inizialmente la stringa type e il numero di tutorials presenti. La stringa risultante è analoga a quella mostrata negli esempi 1 e 2, per il metodo *convertUsers*, con la differenza che al posto degli attributi degli utenti, si hanno quelli dei tutorial

5.5 Package transfer

Il package transfer si occupa della comunicazione con il client.

Costituito dalle classi:

- `mytalk.server.transfer.ListenerTransfer` (vedasi sezione 5.5.1)
- `mytalk.server.transfer.AuthenticationTransfer` (vedasi sezione 5.5.2)
- `mytalk.server.transfer.CallTransfer` (vedasi sezione 5.5.3)
- `mytalk.server.transfer.ChatTransfer` (vedasi sezione 5.5.4)
- `mytalk.server.transfer.FileTransfer` (vedasi sezione 5.5.5)
- `mytalk.server.transfer.RecordMessageTransfer` (vedasi sezione 5.5.6)
- `mytalk.server.transfer.UserTransfer` (vedasi sezione 5.5.7)

5.5.1 `mytalk.server.transfer.ListenerTransfer`

La classe astratta ListenerTransfer implementa WebSocketServerTokenListener e viene estesa da tutte le altre classi dello strato.

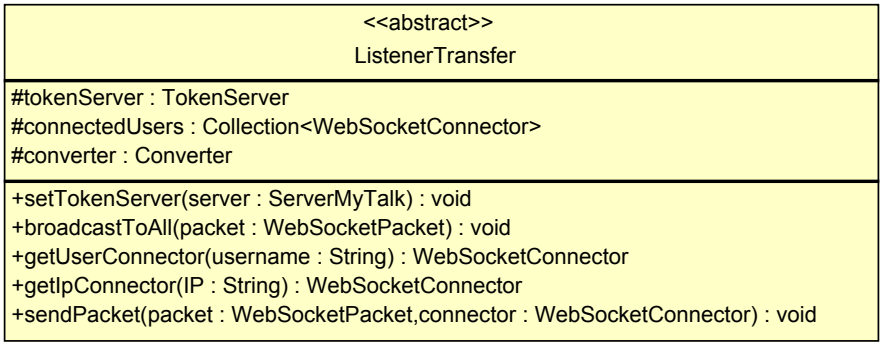

Figura 46: Classe ListenerTransfer

Attributi protetti

- **TokenServer tokenServer:** riferimento al token server di JWebSocket
- **Collection<WebSocketConnector> connectedUsers:** lista dei connettori collegati al server
- **Converter converter:** riferimento alla classe Converter

Metodi pubblici

- **void setTokenServer(ServerMyTalk server):** inizializza *tokenServer*
- **void broadcastToAll(WebSocketPacket packet):** invia il pacchetto in broadcast a tutti gli utenti presenti in *connectedUsers*
- **WebSocketConnector getUserConnector(String username):** ritorna il connettore dell'utente connesso, con il dato username, al sistema. Se non lo trova ritorna null
- **WebSocketConnector getIpConnector(String IP):** ritorna il connettore, avente il dato indirizzo IP, connesso al sistema. Se non lo trova ritorna null
- **void sendPacket(WebSocketPacket packet, WebSocketConnector connector):** invia il dato pacchetto al dato connettore

5.5.2 `mytalk.server.transfer.AuthenticationTransfer`

La classe AuthenticationTransfer si occupa della gestione delle richieste di registrazione o autenticazione degli utenti

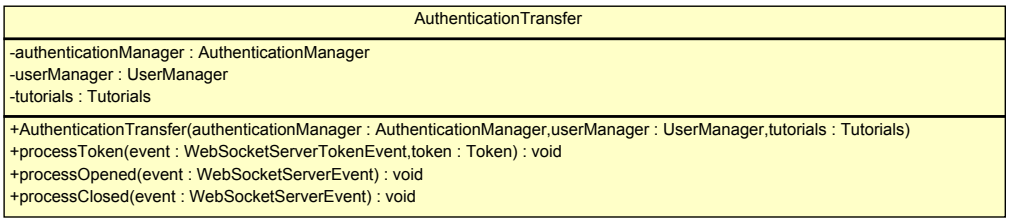

Figura 47: Classe AuthenticationTransfer

Attributi privati

- **AuthenticationManager authenticationManager:** riferimento alla singola istanza della classe AuthenticationManager presente nel server
- **UserManager userManager:** riferimento alla singola istanza della classe UserManager presente nel server
- **Tutorials tutorials:** riferimento ai tutorials presenti nel package shared

Metodi pubblici

- **AuthenticationTransfer(AuthenticationManager authenticationManager, UserManager userManager, Tutorials tutorials):** costruttore della classe, inizializza gli attributi della classe con i parametri ricevuti
- **void processToken(WebSocketServerTokenEvent event, Token token):** gestisce i token in arrivo aventi come tipo:
  - *login*: invoca il metodo *login* della classe AuthenticationManager, usando come parametri lo username e la password, presenti nel token in chiaro, e l'indirizzo IP del connettore da cui riceve il pacchetto. Se l'operazione lancia un'eccezione *Exception* viene catturata e inviata una risposta negativa al client, specificando il tipo di errore, che può essere *Password errata* o *Username errato*. Altrimenti, assegna al connettore lo username dell'utente autenticato, invia una risposta positiva al client e segnala a tutti gli utenti connessi, utilizzando il metodo *broadcastToAll,* la connessione dell'utente inviando un pacchetto di tipo *getContacts,* ottenuto utilizzando il metodo *convertUsers* della classe Converter, e passando come vettore di utenti, un vettore contenente il contatto con il nuovo indirizzo IP
  - *signUp*: invoca il metodo *createUser* della classe AuthenticationManager, usando come parametri i dati presenti nel token, e l'indirizzo IP del connettore da cui riceve il pacchetto. Se l'operazione lancia un'eccezione *Exception* viene catturata e inviata una risposta negativa al client, specificando il tipo di errore, che può essere *Operazione di*

registrazione fallita o *Username utilizzato da un altro utente.* Altrimenti assegna al connettore lo username dell'utente autenticato, invia una risposta positiva al client e segnala a tutti gli utenti connessi, utilizzando il metodo *broadcastToAll,* la connessione del nuovo utente inviando un pacchetto di tipo *getContacts,* ottenuto utilizzando il metodo *convertUsers* della classe Converter

- *getContacts*: invoca il metodo *getAllContacts* della classe AuthenticationManager, per ottenere il vettore degli utenti presenti nel server, e il metodo *convertUsers* della classe Converter per convertire tale lista in stringa JSON, infine restituisce al client il pacchetto contenente tale stringa
- *logout*: invoca il metodo *logout* della classe AuthenticationManager, rimuove dal connettore lo username, se l'operazione va a buon fine, segnala a tutti gli utenti connessi, utilizzando il metodo *broadcastToAll,* la disconnessione dell'utente inviando un pacchetto di tipo *getContacts,* ottenuto utilizzando il metodo *convertUsers* della classe Converter

- **void processOpened(WebSocketServerEvent event):** metodo che si occupa di gestire la connessione di un dispositivo al server. Aggiunge il nuovo connettore a *connectedUsers,* ed invia al nuovo connettore la lista di tutorials, invocando il metodo *convertTutorials*
- **void processClosed(WebSocketServerEvent event):** metodo che si occupa di gestire la disconnessione di un dispositivo dal server. Se non si è fatta l'operazione di *logout* dal client, effettua le operazioni sopraindicate, riguardanti il *logout.* Infine rimuove il connettore da *connectedUsers*

5.5.3 `mytalk.server.transfer.CallTransfer`

La classe CallTransfer si occupa di gestire il traffico di pacchetti per effettuare l'inizializzazione di una comunicazione audio/video.

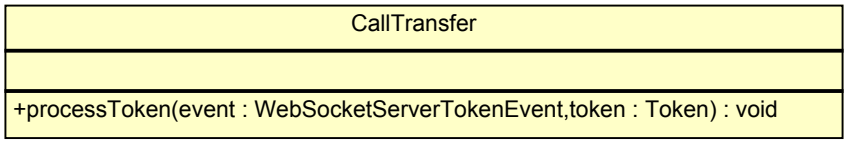

Figura 48: Classe CallTransfer

Metodi pubblici

- **void processToken(WebSocketServerTokenEvent event, Token token):** gestisce dei token in arrivo. Per ogni token che gestisce, invoca il

metodo *getUserConnector*, usando come parametro lo username presente nel token, per trovare il connettore corrispondente. Se non esiste invia al connettore che ha effettuato la richiesta un pacchetto con tipo *answeredCall* avente come risposta *error*. Se il connettore esiste risponde con un pacchetto di tipo diverso a seconda della richiesta:

- *call*: controlla se l'utente è autenticato oppure no e realizza il *WebSocketConnector* adeguato, se l'utente prova ad autenticarsi ma fallisce manda al client il messaggio di errore *Utente non connesso al server*
- *answeredCall:* invia un pacchetto *wspacket*, con tipo *answeredCall* contenente il connettore, e la risposta *true*
- *refuseCall:* invia un pacchetto di tipo *answeredCall*, avente come risposta *false* e il messaggio *Chiamata rifiutata*
- *busy:* invia un pacchetto di tipo *answeredCall*, avente come risposta *false* e il messaggio *Utente occupato in un'altra conversazione*
- *refusecam*: invia un pacchetto di tipo *answeredCall*, avente come risposta *false* e il messaggio *Utente rifiuta di accendere la telecamera*
- *sdp:* invia un pacchetto contenente l'attributo *description* passatogli dal client che ha effettuato la richiesta
- *candidate:* invia un pacchetto contenente il campo *contact* passatogli dal client che ha effettuato la richiesta, che può essere un indirizzo *IP* o un *contact* autenticato
- *endCall:* invia un pacchetto di tipo *endCall* che interrompe la chiamata col connettore selezionato
- *candidateReady:* invia un pacchetto di tipo *candidateReady* che comunica ad un utente che l'altro utente è pronto ad accettare candidati
- *endCallEarly:* invia un pacchetto di tipo *endCallEarly* che interrompe la chiamata col connettore selezionato
- *addConferenceCaller:* invia un pacchetto contenente un connettore da aggiungere ad una chiamata
- *addConferenceAnswer:* invia un pacchetto contenente un connettore da aggiungere ad una chiamata

5.5.4 `mytalk.server.transfer.ChatTransfer`

La classe ChatTransfer si occupa della registrazione dei messaggi di chat.

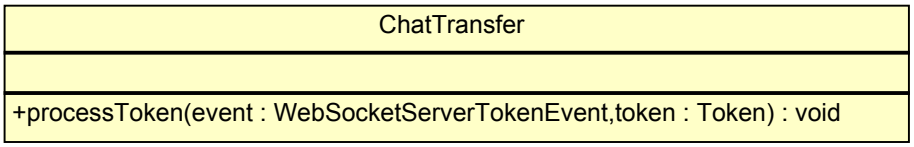

Figura 49: Classe ChatTransfer

Metodi pubblici

- **void processToken(WebSocketServerTokenEvent event, Token token):** gestisce i token in arrivo aventi come tipo *sendText*, invoca il metodo *getUserConnector*, usando come parametro lo username presente nel token, per trovare il connettore corrispondente. Se il connettore non è presente, invia al connettore che ha effettuato la richiesta un pacchetto con tipo *notDelivered,* altrimenti manda al connettore trovato un pacchetto di tipo *sendText*, avente come attributi il messaggio ricevuto in arrivo e lo username del mittente

5.5.5 `mytalk.server.transfer.FileTransfer`

La classe FileTransfer si occupa della gestione dei file inviati. Nonostante tale classe sia stata pianificata, non è stata implementata.

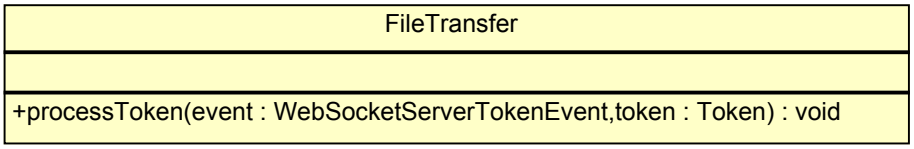

Figura 50: Classe FileTransfer

Metodi pubblici

- **void processToken(WebSocketServerTokenEvent event, Token token):** gestisce i token in arrivo aventi come tipo:
  - *file*: invoca il metodo *getUserConnector*, usando come parametro lo username presente nel token, per trovare il connettore corrispondente. Infine manda al connettore trovato un pacchetto con lo stesso tipo, avente come attributi il file in arrivo e lo username del mittente
  - *refuseFile*: invoca il metodo *getUserConnector*, usando come parametro lo username presente nel token, per trovare il connettore corrispondente. Infine manda al connettore trovato un pacchetto con

tipo *fileRefused*, avente come attributo lo username del mittente del pacchetto

5.5.6 `mytalk.server.transfer.RecordMessageTransfer`

La classe RecordMessageTransfer si occupa della registrazione dei messaggi. Tale classe, seppur pianificata, non è stata implementata

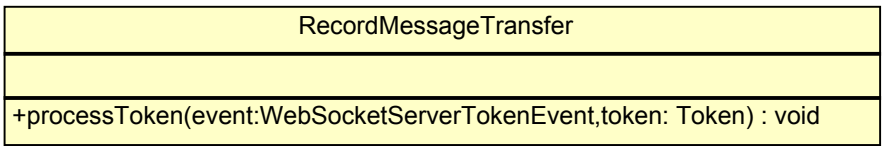

Figura 51: Classe RecordMessageTransfer

Metodi pubblici

- **void processToken(WebSocketServerTokenEvent event, Token token):** gestisce i token in arrivo aventi come tipo:
  - *sendRecord*: invoca il metodo *createMessage* della classe UserManager, per salvare il messaggio nel server, usando come parametri quelli ricevuti. Se l'operazione lancia un'eccezione *Exception* viene catturata e viene inviato al client un pacchetto di tipo *sendRecord*, avente risposta negativa, specificando il tipo di errore, che è *Errore nella registrazione del messaggio*. Altrimenti viene inviato al mittente un messaggio dello stesso tipo avente risposta positiva. Infine se il metodo restituisce il messaggio, e quindi l'utente è connesso, gli invia un pacchetto di tipo *record*, contenente le informazioni ricevute
  - *removeRecord:* invoca il metodo *removeMessage* della classe UserManager per eliminare il messaggio dal server, restituisce al client un pacchetto dello stesso tipo, avente la riuscita o meno dell'operazione

5.5.7 `mytalk.server.transfer.UserTransfer`

La classe UserTransfer si occupa della gestione delle modifiche dei dati dell'utente.

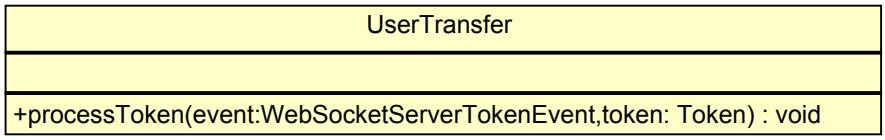

Figura 52: Classe UserTransfer

Metodi pubblici

- **UserTransfer(UserManager userManager):** costruttore della classe, si collega alla istanza esistente di UserManager
- **void processToken(WebSocketServerTokenEvent event, Token token):** gestisce i token in arrivo aventi come tipo:
  - *checkCredentials*: invoca il metodo *checkPassword* di UserManager, passandogli come parametro lo user corrispondente all'utente che ha effettuato la richiesta e la password presente nel pacchetto, salvata in chiaro, e invia al client la risposta ottenuta
  - *changeData:* invoca il metodo di UserManager *setUserData* e verifica che ritorni quanto atteso, in caso contrario lancia un'eccezione, in seguito invoca *setPassword*, sempre di UserManager, per sostituire ai dati salvati nel database quelli ricevuti nel pacchetto e fa in modo che il cambiamento venga mandato in broadcast a tutti gli utenti

6 Tracciamento componenti-requisiti

Componente	Classi	Requisiti
CCLI1	mytalk.client.view.AuthenticationView.signup() mytalk.client.communication.AuthenticationCommunication.signup() mytalk.client.view.AuthenticationView.viewSignup() mytalk.client.view.AuthenticationView.deny()	RUFO 1 RUFO 1.1 RUFO 1.2 RUFO 1.3 RUFO 1.4 RUFO 1.5
	mytalk.client.view.AuthenticationView.connect() mytalk.client.communication.AuthenticationCommunication.checkCredentials()	RUFO 2 RUFO 2.1 RUFO 2.2
	mytalk.client.view.AuthenticationView.disconnect() mytalk.client.communication.AuthenticationCommunication.logout()	RUFO 8
CCLI2	mytalk.client.view.UserDataView.changeData() mytalk.client.communication.UserDataCommunication.changeData() mytalk.client.model.UserModel	RUFF 3 RUFF 3.1 RUFF 3.2 RUFF 3.3 RUFF 3.5
	mytalk.client.view.UserDataView.checkPassword() mytalk.client.communication.UserDataCommunication.checkPassword()	RUFF 3.4
CCLI3	mytalk.client.view.TutorialView.viewTutorial() mytalk.client.communication.TutorialCommunication.getTutorials() mytalk.client.collection.TutorialsCollection mytalk.client.model.TutorialModel mytalk.client.view.TutorialView.next() mytalk.client.view.TutorialView.prev()	RUFF 4 RUFF 4.1
CCLI4	mytalk.client.view.SideView.listContacts() mytalk.client.view.SideView.viewContact() mytalk.client.view.ContactView.view() mytalk.client.communication.ContactsCommunication.fetchContacts() mytalk.client.collection.ContactsCollection.record() mytalk.client.model.ContactModel	RUFO 5 RUFF 5.1 RUFO 6.3
CCLI5	mytalk.client.view.NotificationView.render() mytalk.client.communication.NotificationCommunication.onNotification()	RUFF 7 RUFF 7.1 RUFF 7.4 RUFF 7.7
	mytalk.client.view.NotificationView.acceptCall()	RUFF 7.2
	mytalk.client.view.NotificationView.refuseCall() mytalk.client.communication.NotificationCommunication.refuseCall()	RUFF 7.3
	mytalk.client.view.NotificationView.viewMessage()	RUFF 7.5
	mytalk.client.view.NotificationView.acceptFile()	RUFF 7.8
	mytalk.client.communication.NotificationCommunication.refuseFile()	RUFF 7.9
CCLI6	mytalk.client.view.FunctionsView.call() mytalk.client.communication.CallCommunication.sendCall()	RUFO 6. RUFO 6.1

	mytalk.client.communication.CallCommunication.startCall() mytalk.client.communication.CallCommunication.connect() mytalk.client.communication.CallCommunication.createPeerConnection() mytalk.client.communication.CallCommunication.gotDescription() mytalk.client.communication.CallCommunication.onIceCandidate()	RUFO 6.2 RUFF 6.6 RUFF 6.7 RUFD 6.8 RUFD 6.9 RUFD 6.13
	mytalk.client.view.SideView.callIP()	RUFO 6.1
	mytalk.client.view.FunctionsView.audiocall()	RUFO 6.4
	mytalk.client.view.FunctionsView.videocall()	RUFO 6.5
	mytalk.client.view.CallView.endCall() mytalk.client.communication.CallCommunication.endCall()	RUFO 6.15
CCLI7	mytalk.client.view.RecordMessageView.startRecording() mytalk.client.view.RecordMessageView.stopRecording() mytalk.client.view.RecordMessageView.sendRecordMessage() mytalk.client.communication.RecordMessageCommunication.sendRecordMessage()	RUFF 6.10 RUFF 6.11
	mytalk.client.view.RecordMessageView.viewMessage()	RUFF 7.5
	mytalk.client.view.RecordMessageView.removeMessage() mytalk.client.communication.RecordMessageCommunication.sendRemoveMessage()	RUFF 7.6
CCLI8	mytalk.client.view.ChatView.send() mytalk.client.view.ChatView.putMessages() mytalk.client.communication.ChatCommunication.send() mytalk.client.communication.ChatCommunication.onReceived() mytalk.client.collection.TextMessageCollection.chatSession() mytalk.client.model.TextMessageModel	RUFF 6.12
CCLI9	mytalk.client.view.FileView.sendFile() mytalk.client.communication.FileCommunication.sendFile()	RUFD 6.14 RUFD 6.14.1

CSER1	mytalk.server.transfer.ListenerTransfer.sendPacket() mytalk.server.transfer.ListenerTransfer.broadcastToAll() mytalk.server.transfer.AuthenticationTransfer.processToken() mytalk.server.functionmanager.Converter.convertUsers() mytalk.server.usermanager.AuthenticationManager.createUser() mytalk.server.dao.UserDao.addUser() mytalk.server.dao.UserDaoSQL.addUser()	RUFO 1 RUFO 1.1
	mytalk.server.usermanager.AuthenticationManager.login() mytalk.server.dao.UserDao.checkPassword() mytalk.server.dao.UserDaoSQL.checkPassword() mytalk.server.dao.UserDao.setIP() mytalk.server.dao.UserDaoSQL.setIP()	RUFO 2
	mytalk.server.usermanager.AuthenticationManager.getAllContacts() mytalk.server.shared.UserList.getAllUsers()	RUFO 5 RUFF 5.1
	mytalk.server.transfer.AuthenticationTransfer.processClosed() mytalk.server.usermanager.AuthenticationManager.logout() mytalk.server.dao.UserDao.setIP() mytalk.server.dao.UserDaoSQL.setIP()	RUFO 8

CSER2	mytalk.server.transfer.ListenerTransfer.sendPacket() mytalk.server.transfer.UserTransfer.processToken() mytalk.server.functionmanager.Converter.convertUsers()	RUFF 3
	mytalk.server.usermanager.UserManager.setUserData() mytalk.server.dao.UserDao.setName() mytalk.server.dao.UserDaoSQL.setName() mytalk.server.dao.UserDao.setSurname() mytalk.server.dao.UserDaoSQL.setSurname()	RUFF 3.1 RUFF 3.2
	mytalk.server.usermanager.UserManager.setPassword() mytalk.server.dao.UserDao.setPassword() mytalk.server.dao.UserDaoSQL.setPassword()	RUFF 3.3
CSER3	mytalk.server.transfer.ListenerTransfer.sendPacket() mytalk.server.transfer.CallTransfer.processToken()	RUFO 6 RUFO 6.1 RUFO 6.2 RUFO 6.4 RUFO 6.5 RUFF 6.6 RUFF 6.7 RUFD 6.13 RUFO 6.15 RUFF 7.2 RUFF 7.3
CSER4	mytalk.server.transfer.ListenerTransfer.sendPacket() mytalk.server.transfer.ChatTransfer.processToken() mytalk.server.transfer.FileTransfer.processToken()	RUFF 6.12 RUFD 6.14 RUFF 7.7 RUFF 7.8 RUFF 7.9
CSER5	mytalk.server.transfer.RecordMessageTransfer.processToken() mytalk.server.usermanager.UserManager.createMessage() mytalk.server.functionmanager.Converter.convertMessages() mytalk.server.dao.RecordMessageDao.addMessage() mytalk.server.dao.RecordMessageDaoSQL.addMessage()	RUFF 6.10 RUFF 6.11
	mytalk.server.usermanager.UserManager.getMessages() mytalk.server.dao.RecordMessageDao.getAllMessages() mytalk.server.dao.RecordMessageDaoSQL.getAllMessages()	RUFF 7.5
	mytalk.server.usermanager.UserManager.removeMessage() mytalk.server.dao.RecordMessageDao.addMessage() mytalk.server.dao.RecordMessageDaoSQL.addMessage()	RUFF 7.6
CSER6	mytalk.server.transfer.AuthenticationTransfer.processOpened() mytalk.server.functionmanager.Converter.convertTutorials() mytalk.server.dao.TutorialsDaoSQL.getTutorials() mytalk.server.shared.Tutorials.insert() mytalk.server.shared.Tutorials.getTutorials()	RUFF 4 RUFF 4.1

Tabella 1: Tracciamento tra componenti e requisiti

7 Tracciamento requisiti-componenti

7.1 Tracciamento requisiti - componenti client

Requisito	Componente Client
RUFO 1	mytalk.client.view.AuthenticationView.signup()
RUFO 1.1	mytalk.client.communication.AuthenticationCommunication.signup()
RUFO 1.2	mytalk.client.view.AuthenticationView.viewSignup()
RUFO 1.3	mytalk.client.view.AuthenticationView.deny()
RUFO 1.4	
RUFO 1.5	
RUFO 2	mytalk.client.view.AuthenticationView.connect()
RUFO 2.1	mytalk.client.communication.AuthenticationCommunication.checkCredentials()
RUFO 2.2	
RUFF 3	mytalk.client.view.UserDataView.changeData()
RUFF 3.1	mytalk.client.communication.UserDataCommunication.changeData()
RUFF 3.2	mytalk.client.model.UserModel
RUFF 3.3	
RUFF 3.5	
RUFF 3.4	mytalk.client.view.UserDataView.checkPassword()
	mytalk.client.communication.UserDataCommunication.checkPassword()
RUFF 4	mytalk.client.view.TutorialView.viewTutorial()
RUFF 4.1	mytalk.client.communication.TutorialCommunication.getTutorials()
	mytalk.client.collection.TutorialsCollection
	mytalk.client.model.TutorialModel
	mytalk.client.view.TutorialView.next()
	mytalk.client.view.TutorialView.prev()
RUFO 5	mytalk.client.view.SideView.listContacts()
RUFF 5.1	mytalk.client.view.SideView.viewContact()
	mytalk.client.view.ContactView.view()
	mytalk.client.communication.ContactsCommunication.fetchContacts()
	mytalk.client.collection.ContactsCollection.record()
	mytalk.client.model.ContactModel
RUFO 6	mytalk.client.view.FunctionsView.call()
RUFO 6.1	mytalk.client.communication.CallCommunication.sendCall()
RUFO 6.2	mytalk.client.communication.CallCommunication.startCall()
	mytalk.client.communication.CallCommunication.connect()
	mytalk.client.communication.CallCommunication.createPeerConnection()
	mytalk.client.communication.CallCommunication.onIceCandidate()
	mytalk.client.communication.CallCommunication.gotDescription()
RUFO 6.3	mytalk.client.view.SideView.listContacts()
	mytalk.client.view.SideView.viewContact()
	mytalk.client.view.ContactView.view()
	mytalk.client.communication.ContactsCommunication.fetchContacts()
	mytalk.client.collection.ContactsCollection.record()

	mytalk.client.model.ContactModel
RUFO 6.4	mytalk.client.view.FunctionsView.audiocall()
RUFO 6.5	mytalk.client.view.FunctionsView.videocall()
RUFF 6.6	mytalk.client.view.FunctionsView.call()
RUFF 6.7	mytalk.client.communication.CallCommunication.sendCall()
RUFD 6.8	mytalk.client.communication.CallCommunication.startCall()
RUFD 6.9	mytalk.client.communication.CallCommunication.connect()
RUFD 6.13	mytalk.client.communication.CallCommunication.createPeerConnection()
	mytalk.client.communication.CallCommunication.onIceCandidate()
	mytalk.client.communication.CallCommunication.gotDescription()
RUFF 6.10	mytalk.client.view.RecordMessageView.startRecording()
RUFF 6.11	mytalk.client.view.RecordMessageView.stopRecording()
	mytalk.client.view.RecordMessageView.sendRecordMessage()
	mytalk.client.communication.RecordMessageCommunication.sendRecordMessage()
RUFF 6.12	mytalk.client.view.ChatView.send()
	mytalk.client.view.ChatView.putMessages()
	mytalk.client.communication.ChatCommunication.send()
	mytalk.client.communication.ChatCommunication.onReceived()
	mytalk.client.collection.TextMessageCollection.chatSession()
	mytalk.client.model.TextMessageModel
RUFD 6.14	mytalk.client.view.FileView.sendFile()
RUFD 6.14.1	mytalk.client.communication.FileCommunication.sendFile()
RUFO 6.15	mytalk.client.view.CallView.endCall()
	mytalk.client.communication.CallCommunication.endCall()
RUFF 7	mytalk.client.view.NotificationView.render()
RUFF 7.1	mytalk.client.communication.NotificationCommunication.onNotification()
RUFF 7.2	mytalk.client.view.NotificationView.acceptCall()
RUFF 7.3	mytalk.client.view.NotificationView.refuseCall()
	mytalk.client.communication.NotificationCommunication.refuseCall()
RUFF 7.4	mytalk.client.view.NotificationView.render()
	mytalk.client.communication.NotificationCommunication.onNotification()
RUFF 7.5	mytalk.client.view.NotificationView.viewMessage()
	mytalk.client.view.RecordMessageView.viewMessage()
RUFF 7.6	mytalk.client.view.RecordMessageView.removeMessage()
	mytalk.client.communication.RecordMessageCommunication.sendRemoveMessage()
RUFF 7.7	mytalk.client.view.NotificationView.render()
	mytalk.client.communication.NotificationCommunication.onNotification()
RUFF 7.8	mytalk.client.view.NotificationView.acceptFile()
RUFF 7.9	mytalk.client.communication.NotificationCommunication.refuseFile()
RUFO 8	mytalk.client.view.AuthenticationView.disconnect()
	mytalk.client.communication.AuthenticationCommunication.logout()

Tabella 2: Tracciamento requisiti - componenti client

7.2 Tracciamento requisiti - componenti server

Requisito	Componente Server
RUFO 1	mytalk.server.transfer.ListenerTransfer.sendPacket()
RUFO 1.1	mytalk.server.transfer.ListenerTransfer.broadcastToAll()
	mytalk.server.transfer.AuthenticationTransfer.processToken()
	mytalk.server.functionmanager.Converter.convertUsers()
	mytalk.server.usermanager.AuthenticationManager.createUser()
	mytalk.server.dao.UserDao.addUser()
	mytalk.server.dao.UserDaoSQL.addUser()
RUFO 1.2	-
RUFO 1.3	
RUFO 1.4	
RUFO 1.5	
RUFO 2	mytalk.server.usermanager.AuthenticationManager.login()
	mytalk.server.dao.UserDao.checkPassword()
	mytalk.server.dao.UserDaoSQL.checkPassword()
	mytalk.server.dao.UserDao.setIP()
	mytalk.server.dao.UserDaoSQL.setIP()
RUFO 2.1	-
RUFO 2.2	
RUFF 3	mytalk.server.transfer.ListenerTransfer.sendPacket()
	mytalk.server.transfer.UserTransfer.processToken()
	mytalk.server.functionmanager.Converter.convertUsers()
RUFF 3.1	mytalk.server.usermanager.UserManager.setUserData()
RUFF 3.2	mytalk.server.dao.UserDao.setName()
	mytalk.server.dao.UserDaoSQL.setName()
	mytalk.server.dao.UserDao.setSurname()
	mytalk.server.dao.UserDaoSQL.setSurname()
RUFF 3.3	mytalk.server.usermanager.UserManager.setPassword()
	mytalk.server.dao.UserDao.setPassword()
	mytalk.server.dao.UserDaoSQL.setPassword()
RUFF 3.5	-
RUFF 3.4	
RUFF 4	mytalk.server.transfer.AuthenticationTransfer.processOpened()
RUFF 4.1	mytalk.server.functionmanager.Converter.convertTutorials()
	mytalk.server.dao.TutorialsDaoSQL.getTutorials()
	mytalk.server.shared.Tutorials.insert()
	mytalk.server.shared.Tutorials.getTutorials()
RUFO 5	mytalk.server.transfer.ListenerTransfer.sendPacket()
RUFF 5.1	mytalk.server.transfer.ListenerTransfer.broadcastToAll()
	mytalk.server.usermanager.AuthenticationManager.getAllContacts()
	mytalk.server.shared.UserList.getAllUsers()
RUFO 6	mytalk.server.transfer.ListenerTransfer.sendPacket()
RUFO 6.1	mytalk.server.transfer.CallTransfer.processToken()

RUFO 6.2	
RUFO 6.3	-
RUFO 6.4	mytalk.server.transfer.ListenerTransfer.sendPacket() mytalk.server.transfer.CallTransfer.processToken()
RUFO 6.5	
RUFF 6.6	
RUFF 6.7	
RUFD 6.8	-
RUFD 6.9	
RUFD 6.13	
RUFF 6.10	mytalk.server.transfer.RecordMessageTransfer.processToken() mytalk.server.usermanager.UserManager.createMessage() mytalk.server.functionmanager.Converter.convertMessages() mytalk.server.dao.RecordMessageDao.addMessage() mytalk.server.dao.RecordMessageDaoSQL.addMessage()
RUFF 6.11	
RUFF 6.12	mytalk.server.transfer.ListenerTransfer.sendPacket() mytalk.server.transfer.ChatTransfer.processToken()
RUFD 6.14	mytalk.server.transfer.ListenerTransfer.sendPacket() mytalk.server.transfer.FileTransfer.processToken()
RUFD 6.14.1	-
RUFO 6.15	mytalk.server.transfer.ListenerTransfer.sendPacket() mytalk.server.transfer.CallTransfer.processToken()
RUFF 7	-
RUFF 7.1	
RUFF 7.2	mytalk.server.transfer.ListenerTransfer.sendPacket() mytalk.server.transfer.CallTransfer.processToken()
RUFF 7.3	
RUFF 7.4	-
RUFF 7.5	mytalk.server.usermanager.UserManager.getMessages() mytalk.server.dao.RecordMessageDao.getAllMessages() mytalk.server.dao.RecordMessageDaoSQL.getAllMessages()
RUFF 7.6	mytalk.server.usermanager.UserManager.removeMessage() mytalk.server.dao.RecordMessageDao.addMessage() mytalk.server.dao.RecordMessageDaoSQL.addMessage()
RUFF 7.7	mytalk.server.transfer.ListenerTransfer.sendPacket() mytalk.server.transfer.FileTransfer.processToken()
RUFF 7.8	
RUFF 7.9	
RUFO 8	mytalk.server.transfer.AuthenticationTransfer.processClosed() mytalk.server.usermanager.AuthenticationManager.logout() mytalk.server.dao.UserDao.setIP() mytalk.server.dao.UserDaoSQL.setIP()

Tabella 3: Tracciamento requisiti - componenti server