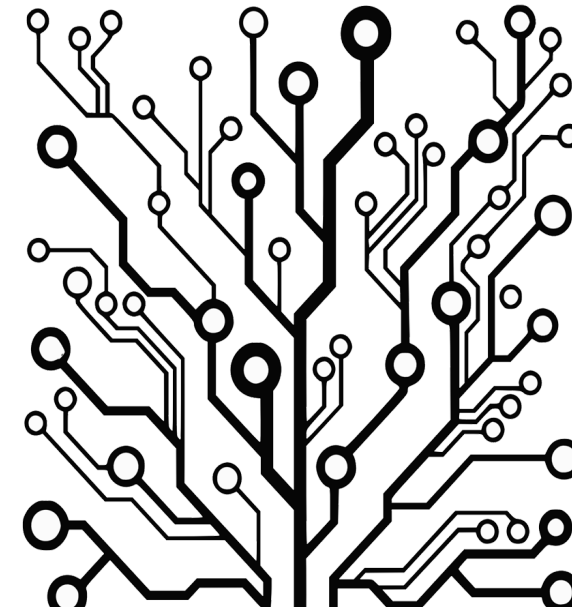


Corso di Laurea Magistrale in
Ingegneria Informatica

Elaborato del corso di *Applicazioni Telematiche*

Secure Messaging

Documentazione



Parricelli Francesco M63/720
Strofaldi Aldo M63/728
Pirozzi Luca M63/744

Anno Accademico 2017/2018

Indice

	Indice	2
1	INTRODUZIONE	3
2	DOCUMENTAZIONE	4
2.1	Documentazione Server Tomcat	4
2.1.1	Diagramma delle Classi	4
2.1.1.1	Package: access control	4
2.1.1.2	Package: authentication	5
2.1.1.3	Package: authorization	7
2.1.1.4	Package: bcrypt	7
2.1.1.5	Package: DAO	8
2.1.1.6	Package: entity	8
2.1.1.7	Package: exception	9
2.1.1.8	Package: keystore	11
2.1.1.9	Package: keystore.rsaKeystore	11
2.1.1.10	Package: registration	11
2.1.1.11	Package: twosteps	11
2.1.1.12	Package: utility	13
2.1.1.13	Package: utility.database	13
2.1.1.14	Package: utility.mail	13
2.1.1.15	Package: utility.network	13
2.1.1.16	Package: webfilter	14
2.1.1.17	Package: websocket	14
2.1.1.18	Package: xml.registration	15
2.2	Sequence Diagrams	15
2.2.1	Server Interaction (Login)	15
2.2.1.1	Offer Interaction	17
2.2.1.2	Answer Interaction	19
2.2.1.3	ICE Candidates Interaction	21
2.2.1.4	Inactivity Timeout Interaction	23
2.2.1.5	Contact List Filter	23

1 Introduzione

Secure Messaging, elaborato del corso di Applicazioni Telematiche, è un'applicazione per la messaggistica sicura, in grado cioè di garantire confidenzialità, sviluppata cercando di renderla il più possibile simile ad una reale.

L'applicazione è strutturata secondo l'architettura Client/Server: nello specifico il Web Server utilizzato è stato Apache Tomcat 9.0, contenente Java Servlet e un modulo NodeJS per la gestione della segnalazione mentre il client è di tipo web, sviluppato facendo uso di HTML, Javascript/JQuery e AJAX, CSS.

Si è inoltre fatto uso di altre tecnologie e linguaggi come ad esempio XACML per il controllo accessi, JSON/XML come formato di interscambio dati, SQL per il database.

Oltre a implementare la comunicazione crittografata end-to-end facendo uso di WebRTC così come da specifica, sono stati introdotti ulteriori meccanismi che ci aspetteremmo di trovare solo in sistemi effettivamente rilasciati sul mercato: ciò risulta essere il risultato di un percorso di studi che ci ha portato a considerare lo sviluppo di un sistema software o embedded come qualcosa che vada oltre la semplice considerazione di correttezza e prestazioni, ma che include anche altri fattori come robustezza, confidenzialità e integrità.

Tenendo sempre in considerazione i vincoli di tempo e di sviluppatori, sono stati affrontati problemi tipici di applicazioni reali come ad esempio la protezione dell'account, che si è tradotta nella limitazione dei tentativi di accesso all'account (fino al blocco IP) per dispositivo, in modo da frustrare tentativi di attacchi brute-force e autenticazione a due passi. Le password, così come dalle linee guida di numerose organizzazioni, sono state criptate (con salt autogenerato) mediante algoritmo bcrypt. Le richieste contengono token che sono limitati nel numero di utilizzi, caratterizzati da un timestamp (e relativa scadenza), associati ad un IP e firmati dal server per evitare attacchi comuni come il replay attack.

Uno sguardo attento è stato rivolto alla considerazione di ogni scenario possibile facendo uso del client web, cercando di garantire che i dati immessi dall'utente siano sempre ben formati alla sorgente (mail, campi numerici, ecc.), garantendo robustezza e alleggerendo il carico del server.

2 Documentazione

2.1 Documentazione Server Tomcat

In questa sezione vengono mostrati i diagrammi UML relativamente alla realizzazione del Server Tomcat, su cui sono installate le Servlet per la gestione delle politiche di login e registrazione, nonché l'interfacciamento con la base di dati e con il mail server.

2.1.1 Diagramma delle Classi

Di seguito viene riportato il diagramma delle classi di Secure Messaging: piuttosto che limitarci ad allegare un solo diagramma, troppo complesso e poco informativo, si riportano le classi per package e relazioni intra ed inter-package; in questo modo abbiamo suddiviso il diagramma delle classi originali in porzioni sufficientemente piccole da poter essere analizzate con profitto ma sufficientemente grandi da essere significative, che nell'insieme permette ad un programmatore la giusta documentazione per individuare struttura e dipendenza delle classi con l'adeguata granularità di informazioni. Per ragioni di semplicità sono state omesse le relazioni al package delle eccezioni, da cui praticamente tutti i package dipendono.

2.1.1.1 Package: **access control**

Contiene la classe che si occupa delle operazioni per prevenire o individuare le intrusioni in un account, aggiornare i conteggi dei login falliti e bloccare i tentativi di accesso.

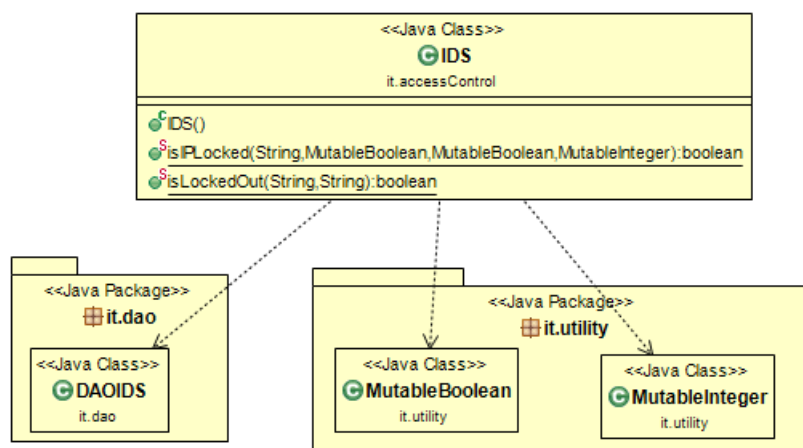


Figura 1 – Package Access Control

2.1.1.2 Package: **authentication**

Appartengono a questo package la servlet di autenticazione e la relativa logica (disaccoppiata dalla servlet per il principio della separazione degli interessi). Il package è visualizzato in figura 2.

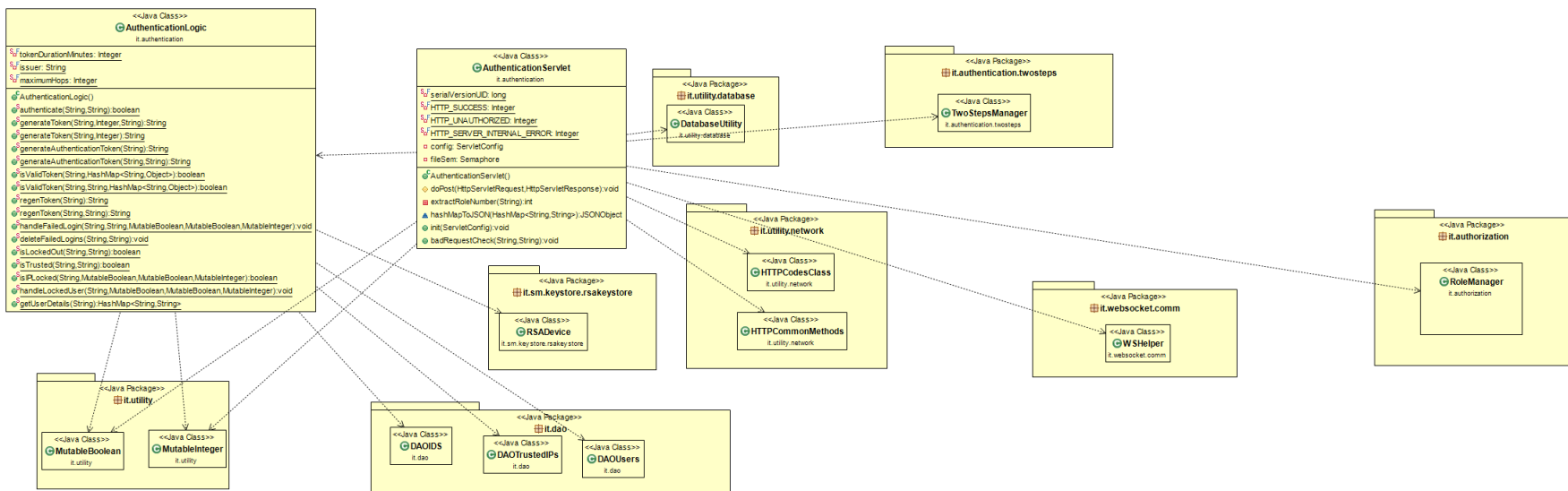


Figura 2 – Package Authentication

2.1.1.3 Package: authorization

Le classi di questo package si occupano di determinare se l'utente è autorizzato ad effettuare una determinata operazione. In particolare, contiene un filtro (ContactListFilter) che si occupa di filtrare le richieste che vengono fatte verso le diverse contact-lists disponibili, ed accettarle in base al rispetto delle policy XACML previste.

Il package è visualizzato in figura 3.

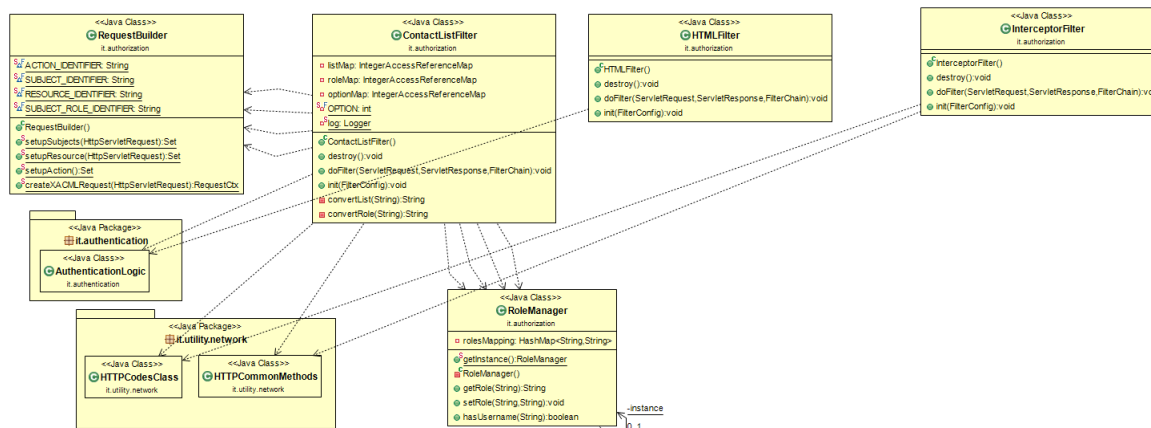


Figura 3 – Package Authorization

2.1.1.4 Package: bcrypt

Package importato da una libreria OTS che implementa hashing/salting delle password con algoritmo bcrypt (utilizzato, fra gli altri, da OpenBSD). Il package è visualizzato in figura 4.

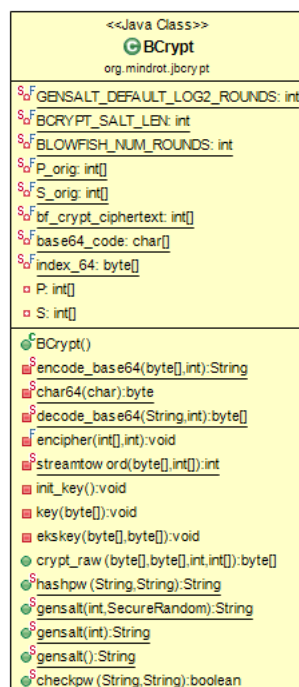


Figura 4 – Package Bcrypt

2.1.1.5 Package: DAO

Questo package rappresenta il ponte tra i dati del database e l'applicazione. Sono le classi a cui le altre si rivolgono per operare su dati che altrimenti sarebbero da prelevare direttamente dal database.

Il package è visualizzato in figura 5.

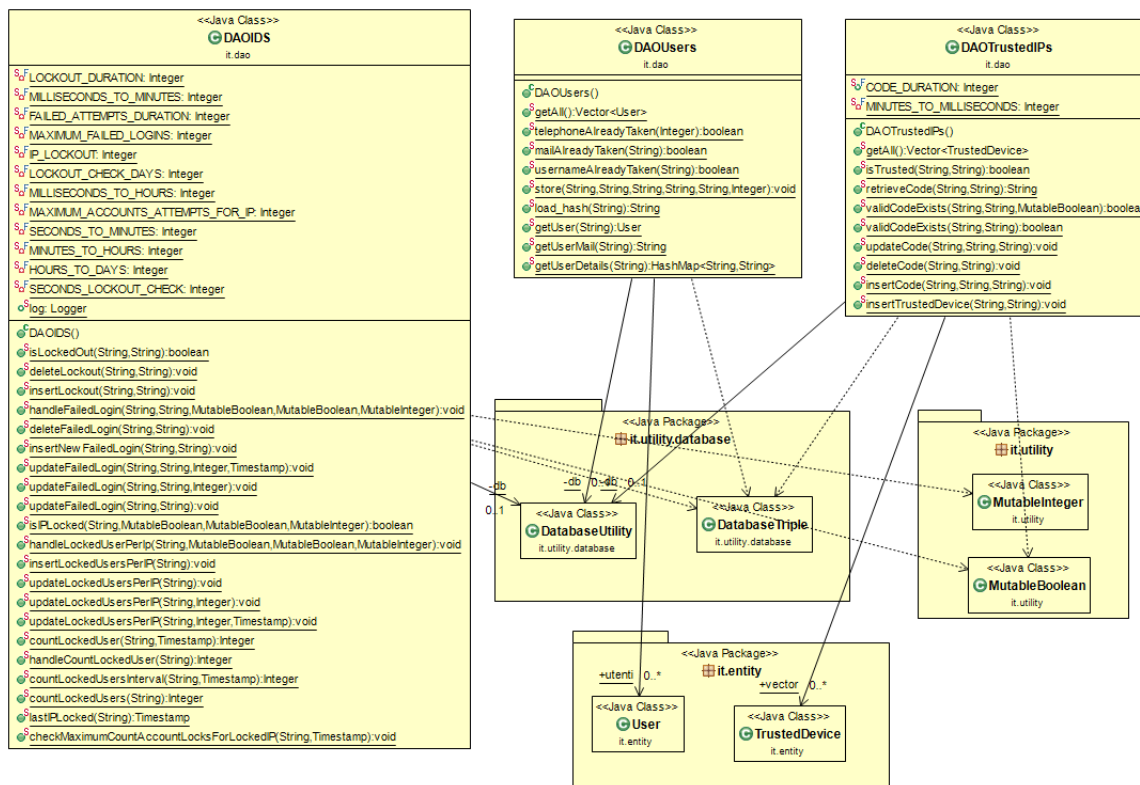


Figura 5 – Package DAO

2.1.1.6 Package: entity

Costituiscono una rappresentazione Object-oriented che effettua il mapping rispetto alla rappresentazione del db.

Il package è visualizzato in figura 6.

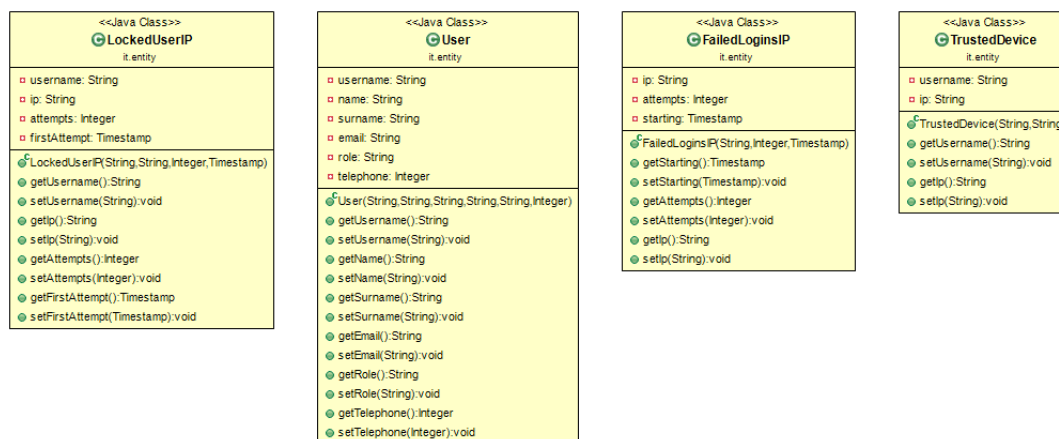


Figura 6 – Package Entity

2.1.1.7 Package: **exception**

Raccoglie tutte le eccezioni che possono essere lanciate. A sua volta è diviso in sotto-package che non vengono rappresentati per semplicità.

Il package è visualizzato in figura 7, a pagina successiva.

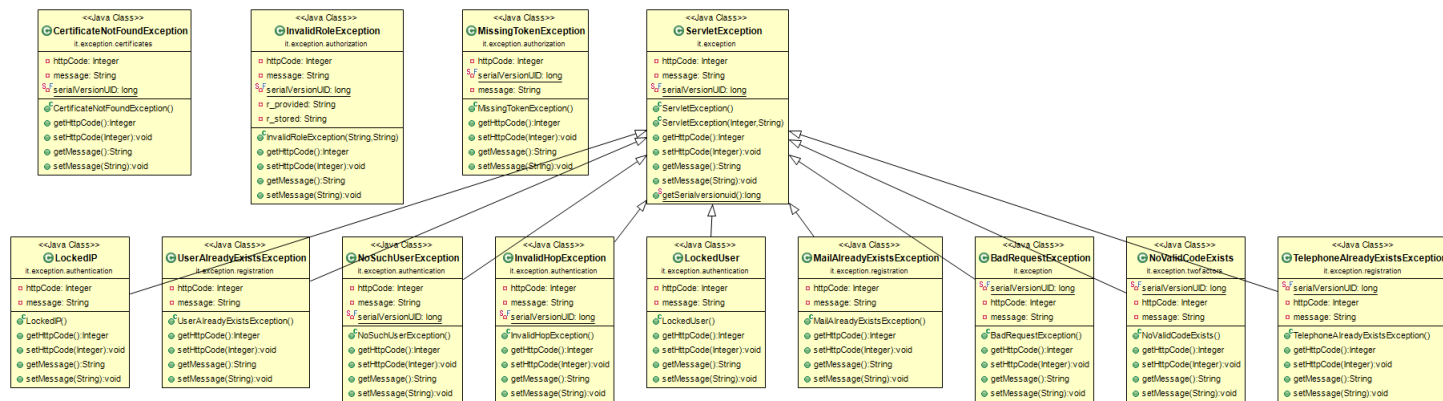


Figura 7 – Package Exception

2.1.1.8 Package: **keystore**

Contiene la semplice interfaccia che ci aspetteremmo da un'entità crittografica: la capacità di criptare e decriptare bytes. Implementeranno l'interfaccia tutte quelle classi che si occuperanno di crittografare dati (implementando gli specifici algoritmi di cui MyKeystore è indipendente).

Il package è visualizzato in figura 8.

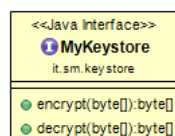


Figura 8 – Package Keystore

2.1.1.9 Package: **keystore.rsaKeystore**

Package innestato nel precedente che rappresenta una crittografia specifica, in questo caso di tipo RSA. Simula le caratteristiche di una smart-card (di cui avremmo bisogno per una crittografia sicura) in software, e usa le chiavi private e pubblica per codificare e decodificare: la privata viene usata per codificare i dati da mandare al client, la pubblica quelli che solo il server può leggere, come la chiave AES utilizzata per codificare i parametri di accesso al db e al mail server.

Il package è visualizzato in figura 9.

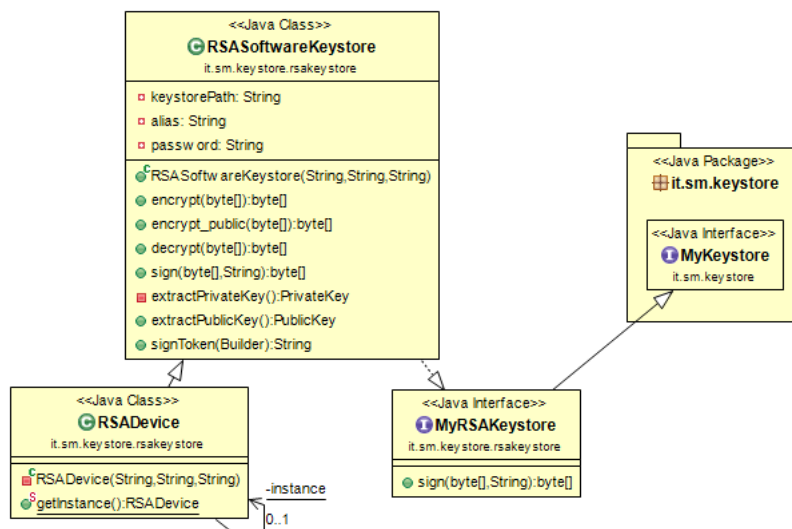


Figura 9 – Package keystore.rsaKeystore

2.1.1.10 Package: **registration**

Appartengono a questo package la servlet e la logica di registrazione.

Il package è visualizzato in figura 10.

2.1.1.11 Package: **twosteps**

Di questo package fanno parte le classi che si occupano dell'autenticazione in due passi; la classe `TwoStepsServlet` rappresenta la servlet a cui rivolgersi per l'invio del codice, ed effettua chiamate a `TwoStepsLogic` per ciò che concerne la logica delle operazioni, mentre `TwoStepsManager` si occupa dell'invio della mail nel caso in cui fosse richiesta.

Il package è visualizzato in figura 11.

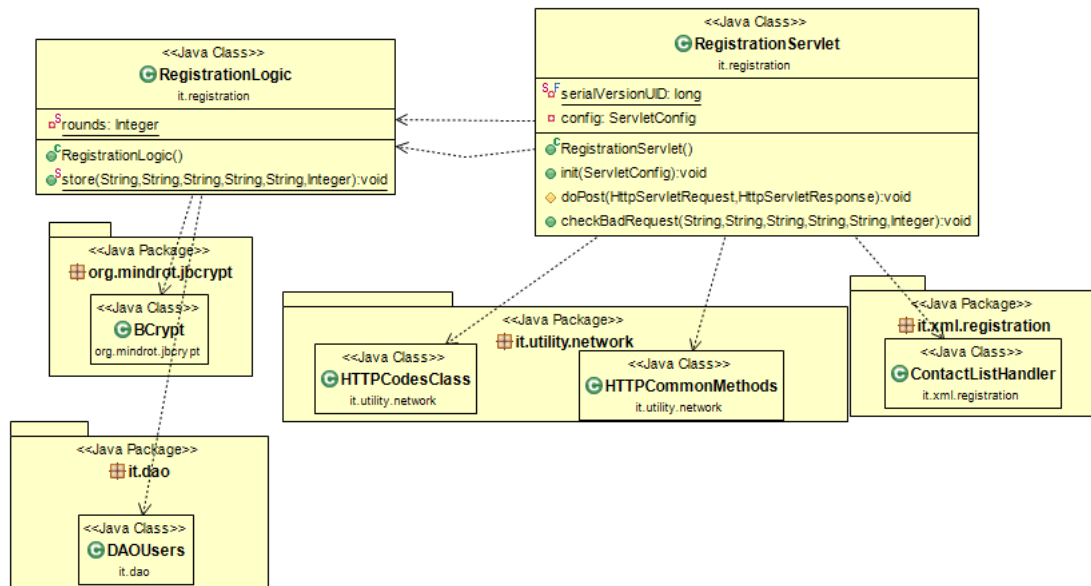


Figura 10 – Package Registration

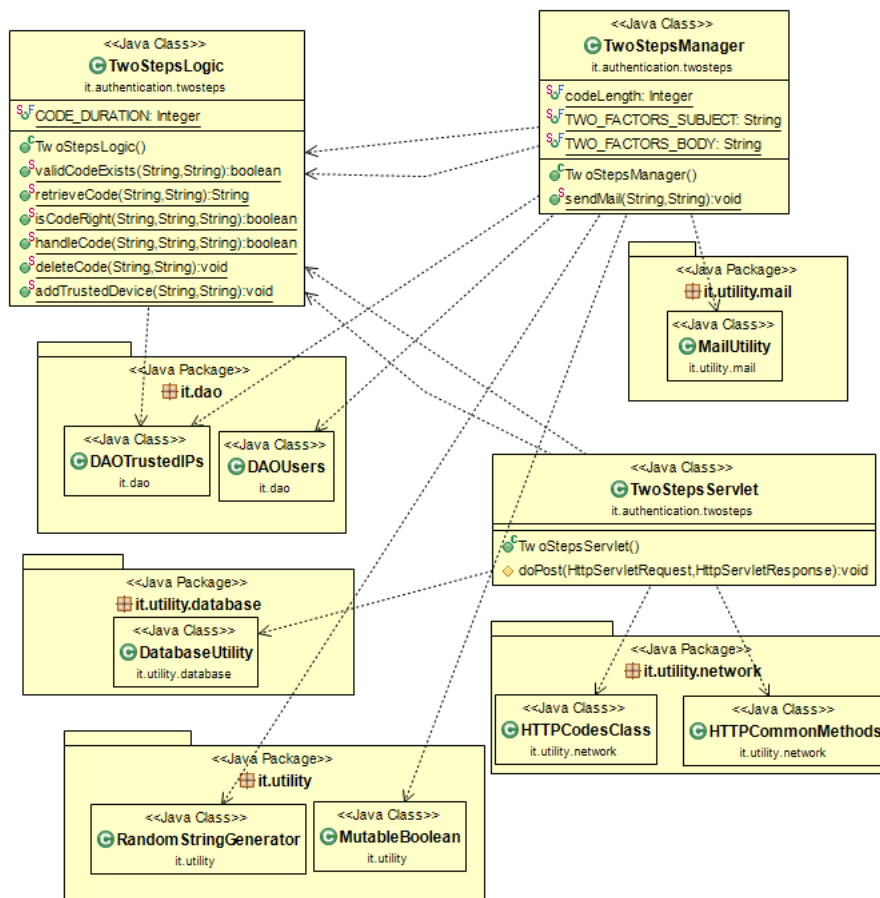


Figura 11 – Package Twosteps

2.1.1.12 Package: **utility**

Contiene diverse classi di utilità, utilizzate per risolvere in modo semplice alcuni semplici compiti, come la generazione del codice per l'autenticazione a due step (RandomStringGenerator).

Il package è visualizzato in figura 12.

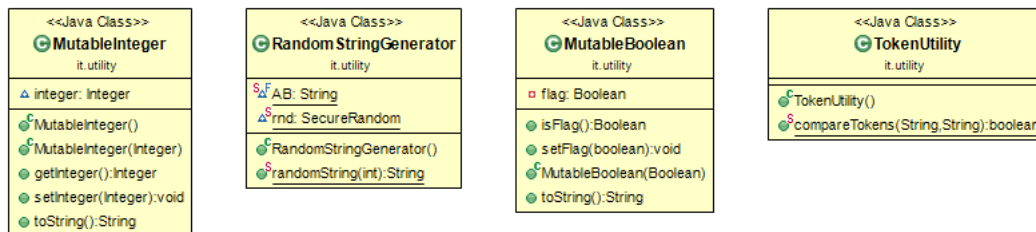


Figura 12 – Package Utility

2.1.1.13 Package: **utility.database**

A questo package appartengono delle classi di utilità create per rendere più semplice alcune operazioni di routine sul database.

Il package è visualizzato in figura 13.

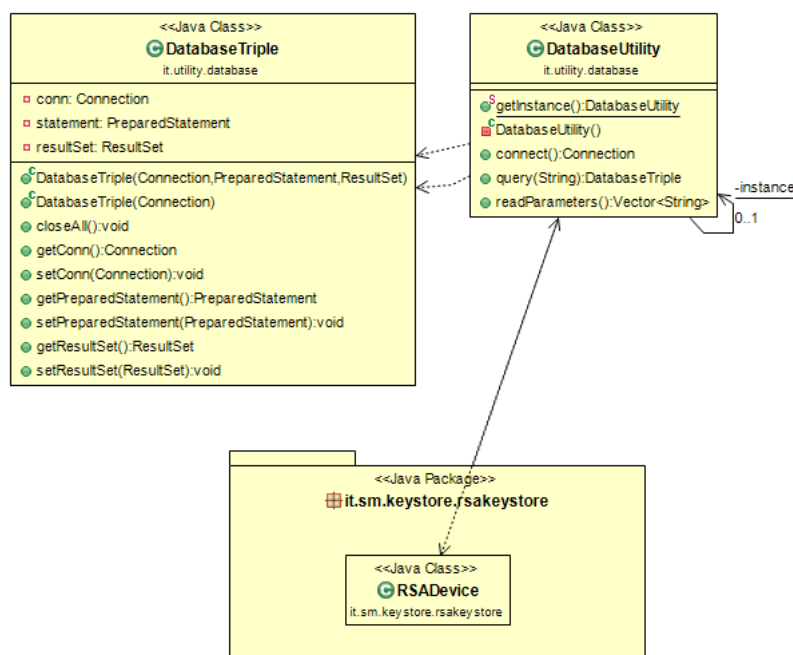


Figura 13 – Package utility.database

2.1.1.14 Package: **utility.mail**

Contiene la classe per l'interfacciamento al mail server.

Il package è visualizzato in figura 14.

2.1.1.15 Package: **utility.network**

Contiene la classe per l'interfacciamento al mail server.

Il package è visualizzato in figura 15.

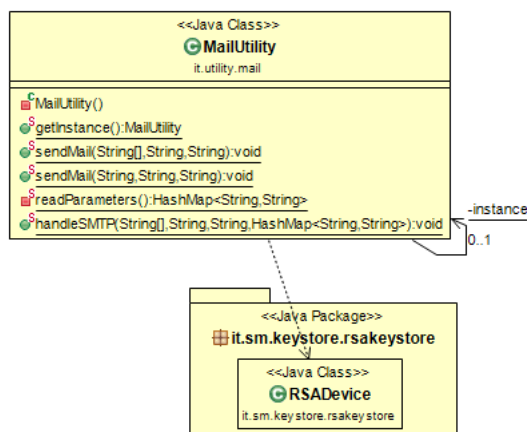


Figura 14 – Package utility.mail

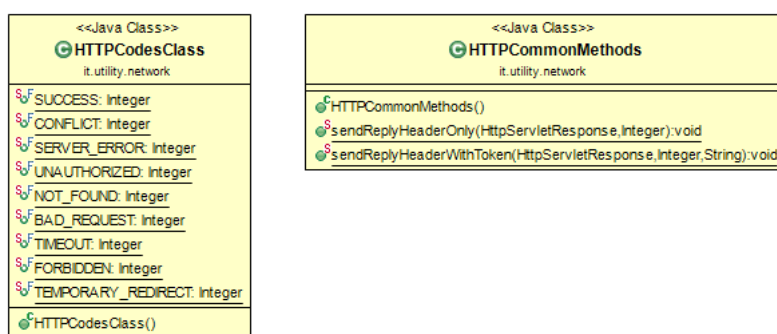


Figura 15 – Package utility.network

2.1.1.16 Package: **webfilter**

Le classi di questo package risolvono diversi problemi come il Cross Origin Resource Sharing. Il package è visualizzato in figura 16.

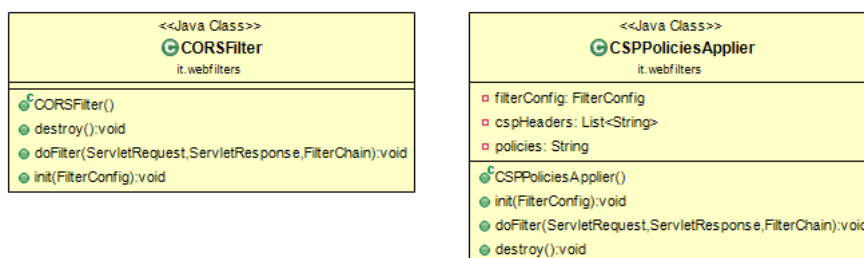


Figura 16 – Package webfilter

2.1.1.17 Package: **websocket**

Questo package contiene le classi per la gestione delle Web Socket. In particolare, contiene classi che consentono l'interfacciamento con il server nodeJS e la comunicazione dei dati relativi agli utenti che hanno correttamente effettuato la procedura di login. Infatti, nel momento in cui un utente effettua il login presso il server principale, le sue informazioni saranno inoltrate anche al server nodeJS: per evitare la falsificazione delle informazioni che vengono inviate al server nodeJS, sono state predisposte alcune misure di sicurezza con cui il server nodeJS è in grado di capire se il messaggio ricevuto proviene effettivamente dal server principale o meno: per ulteriori dettagli, si rimanda ai sequence diagram relativi alla fase di login.

Il package è visualizzato in figura 17.

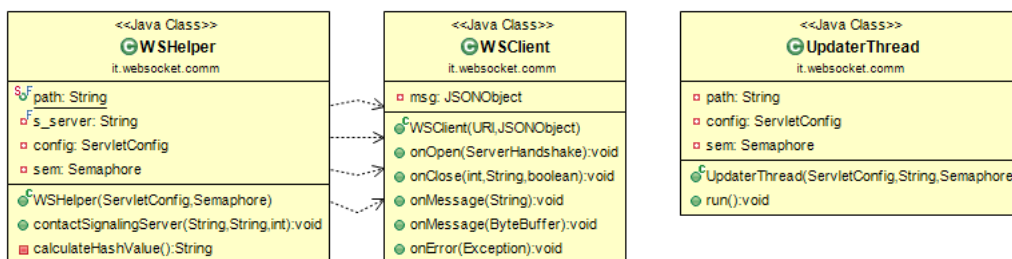


Figura 17 – Package websocket

2.1.1.18 Package: **xml.registration**

Il package è visualizzato in figura 18.

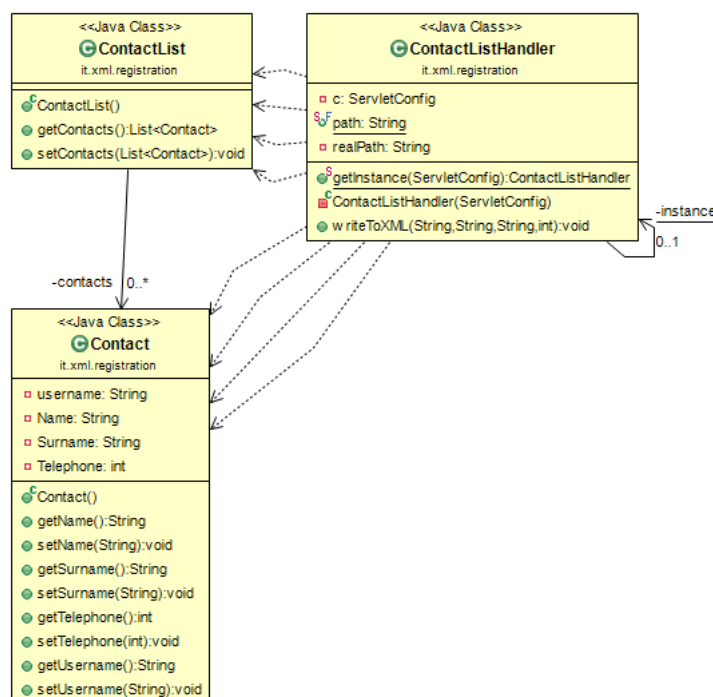


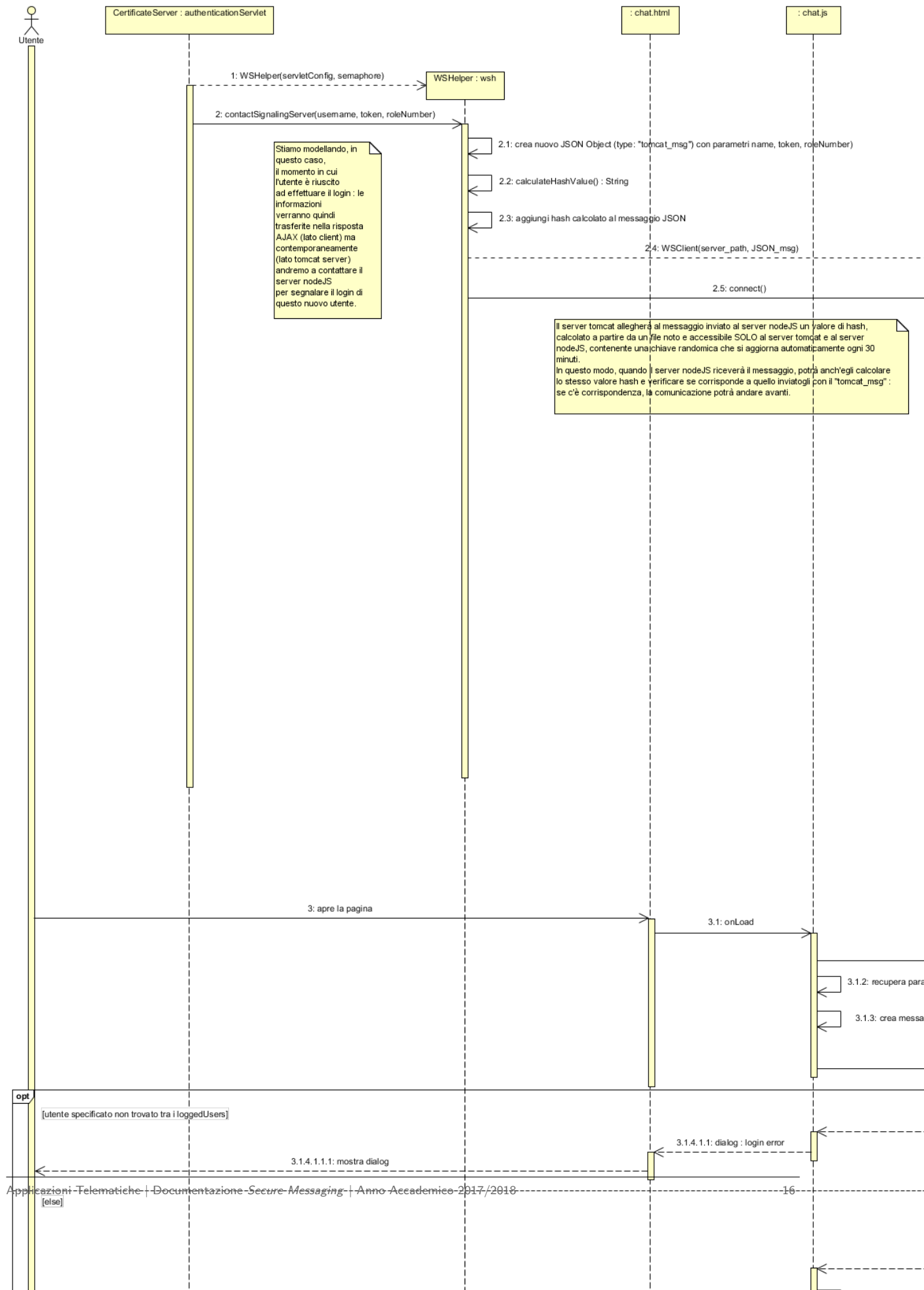
Figura 18 – Package websocket

2.2 Sequence Diagrams

In questa sezione vengono mostrati e commentati i sequence diagram realizzati per le funzionalità più significative dell'applicazione realizzata.

2.2.1 Server Interaction (Login)

Di seguito viene riportato il sequence diagram che descrive l'interazione che avviene tra il server NodeJS e il server Tomcat all'atto del login.



Quando un nuovo utente effettua correttamente il login presso la pagina login.html, accadono due cose:

- Le informazioni relative all'utente loggato vengono trasferite (come AJAX response) al client, che le conserva in un sessionStorage;
- Il server Tomcat si occupa di contattare, mediante websocket, il server di segnalamento (nodeJS) per notificare il login di un nuovo utente.

Nel nostro caso, siamo interessati a modellare il secondo scenario.

Quando questo accade infatti, il server Tomcat aprirà una websocket verso il server nodeJS, passandogli le informazioni relative al nuovo utente loggato e, in aggiunta, un valore di hash: questo valore viene calcolato a partire da un file noto **esclusivamente** al server Tomcat e al server nodeJS (contenente una chiave randomica generata ogni 30 minuti): in questo modo, nel momento in cui il server nodeJS riceverà le informazioni sul login effettuato, potrà calcolare l'hash a partire dallo stesso file e verificarne la validità.

In caso di corrispondenza, il server nodeJS si occuperà di inserire l'utente specificato tra i cosiddetti *loggedUsers*, ovvero coloro che hanno correttamente eseguito la procedura di login presso il server principale.

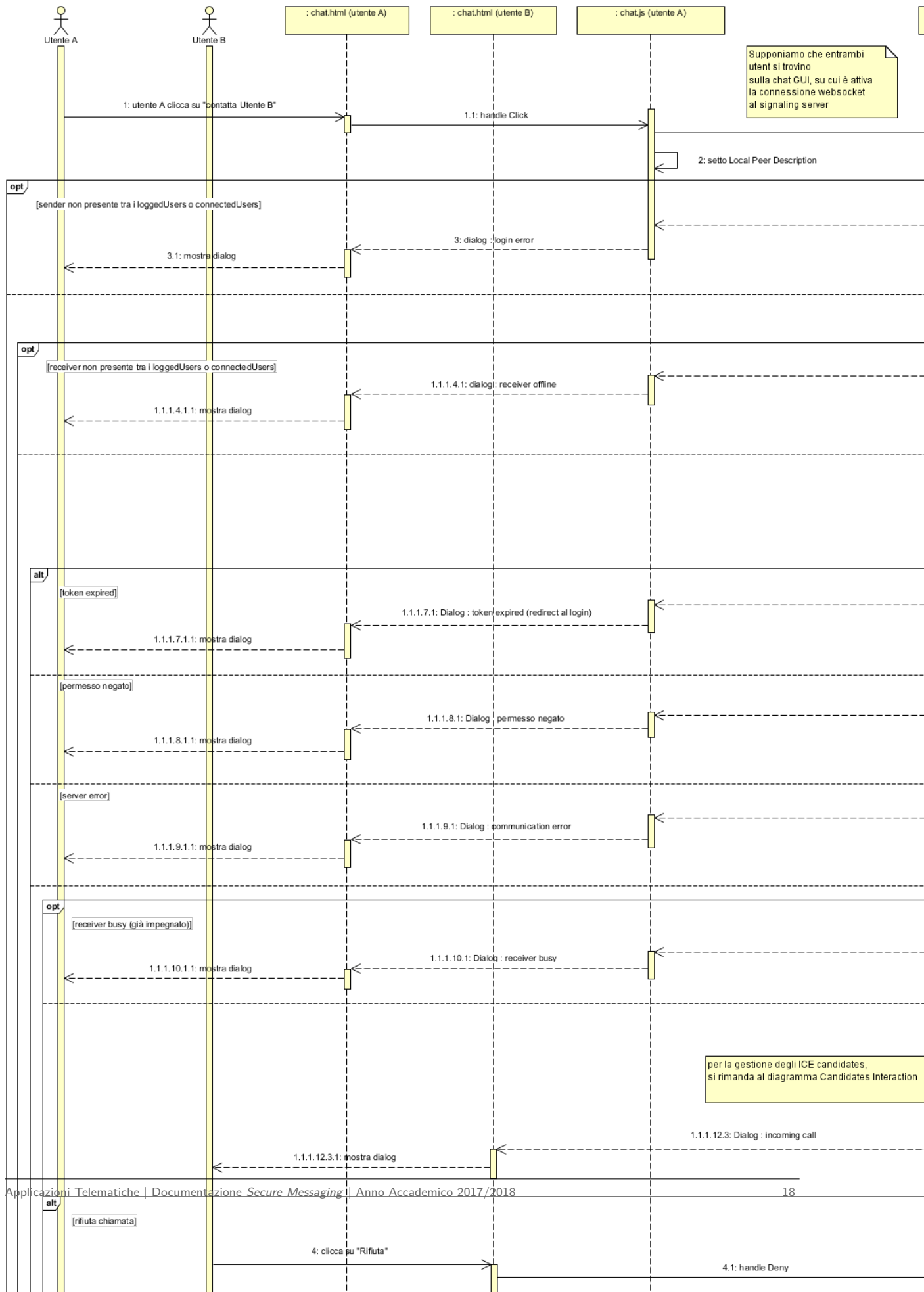
In questo modo, evitiamo che un qualsiasi utente possa contattare il server nodeJS e spacciarsi per un utente loggato.

Quando poi l'utente, accedendo alla pagina chat.html, si conatterà al signaling server (aprendo una websocket verso di esso), comunicherà a quest'ultimo le informazioni che ha ottenuto in fase di login (conservate nel sessionStorage) in modo che il server nodeJS potrà verificare se queste informazioni corrispondono effettivamente ad un utente presente tra i *loggedUsers* (loggato correttamente): in caso affermativo, l'utente sarà inserito nei *connectedUsers*, ovvero gli utenti con una connessione attiva verso il signaling server.

Per ulteriori dettagli si rimanda alla documentazione interna del server nodeJS.

2.2.1.1 Offer Interaction

Il sequence diagram mostra l'interazione che si viene a realizzare ogniqualvolta un utente decide di inviare una *offer* verso un altro utente, con l'intenzione di configurare e avviare una chat.



Se supponiamo di avere un utente A che invia un messaggio offer destinato all'utente B, allora il server nodeJS, alla ricezione di tale messaggio, si occuperà di:

- verificare che l'utente A sia loggato e connesso (ovvero: presente tra `loggedUsers` e `connectedUsers`);
- verificare che l'utente B sia loggato e connesso (ovvero: presente tra `loggedUsers` e `connectedUsers`);
- verificare che l'access token di A sia valido;
- verificare che A abbia il permesso di parlare con B (contattando il server tomcat e usando policy XACML);
- verificare che B non sia già impegnato in un'altra chat.

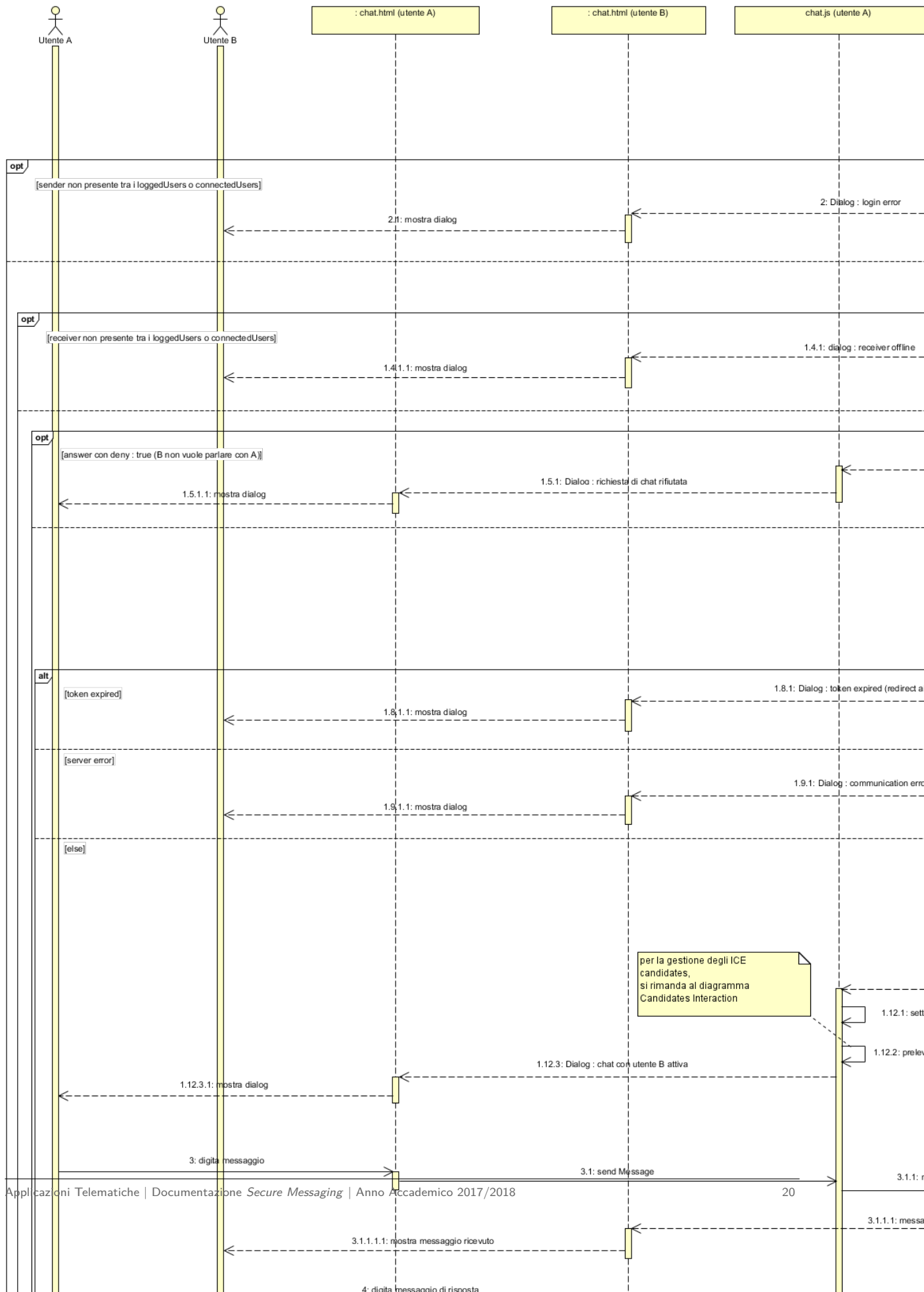
Quando una di queste condizioni non dovesse essere verificata, il server nodeJS si occuperà di contattare il mittente (utente A) con un messaggio di stato opportuno.

Inoltre, se tutte le verifiche di cui sopra vanno a buon fine, il server nodeJS si occuperà di recapitare un messaggio offer verso l'utente B, che permetterà a quest'ultimo di configurare la comunicazione con l'utente A: in particolare, l'utente B potrà in questo modo configurare la sua *LocalPeerDescription* (locale) e la *RemotePeerDescription* (legata all'utente A, e ottenuta con le informazioni contenute nella offer).

Per ulteriori dettagli si rimanda alla documentazione interna del server nodeJS.

2.2.1.2 Answer Interaction

Di seguito si riporta il sequence diagram che mostra l'interazione tra utente A e utente B nel momento in cui vi è l'invio di una *answer*: supponendo che l'utente A voglia chiamare l'utente B, e che l'invio dell'offer sia avvenuto correttamente verso l'utente B, andremo a descrivere la comunicazione che avviene nel momento in cui B decide di inviare una *answer* ad A, tramite il signaling server.



Si nota che, in tal caso, quando l'utente B riceve l'offer da parte dell'utente A, si possono verificare due situazioni:

- l'utente B *rifiuta* di parlare con l'utente A;
- l'utente B *accetta* di parlare con l'utente A.

Nel primo caso, l'utente B invierà al signaling server un messaggio answer contenente un campo *deny*, con il quale il server nodeJS capirà che l'utente B ha rifiutato la chiamata e potrà quindi avvertire l'utente A di tale situazione.

Nel secondo caso, il server nodeJS riceverà un messaggio answer da parte dell'utente B, che dovrà poi essere recapitato ad A dopo aver verificato le stesse condizioni di prima.

In particolare, il server nodeJS si occuperà di:

- verificare che l'utente A sia loggato e connesso (ovvero: presente tra `loggedUsers` e `connectedUsers`)
- verificare che l'utente B sia loggato e connesso (ovvero: presente tra `loggedUsers` e `connectedUsers`)
- verificare che l'access token di B sia valido
- verificare che l'utente B non abbia rifiutato di rispondere (in altri termini: se A chiama B, B può decidere di non rispondere)
- verificare che A abbia il permesso di parlare con B (contattando il server tomcat e usando policy XACML)

Nel caso in cui una di queste condizioni non dovesse essere verificata, l'utente B (ed eventualmente l'utente A) saranno opportunamente avvertiti mediante appositi messaggi di stato.

Se invece la procedura va a buon fine, il signaling server si occuperà di inoltrare il messaggio di answer verso l'utente A: questo consentirà all'utente A di configurare sia la propria *LocalPeerDescription* (locale ad A) sia la *RemotePeerDescription* (legata all'utente B e ottenuta con le informazioni contenute nella answer).

In questo modo (una volta avvenuta anche la configurazione degli ICE candidates) la connessione è finalmente stabilita, e quindi utente A e utente B possono iniziare lo scambio di messaggi.

Per ulteriori dettagli si rimanda alla documentazione interna del server nodeJS.

2.2.1.3 ICE Candidates Interaction

Il seguente sequence diagram illustra il meccanismo con cui avviene la comunicazione e lo scambio degli ICE candidates tra l'utente A e l'utente B.

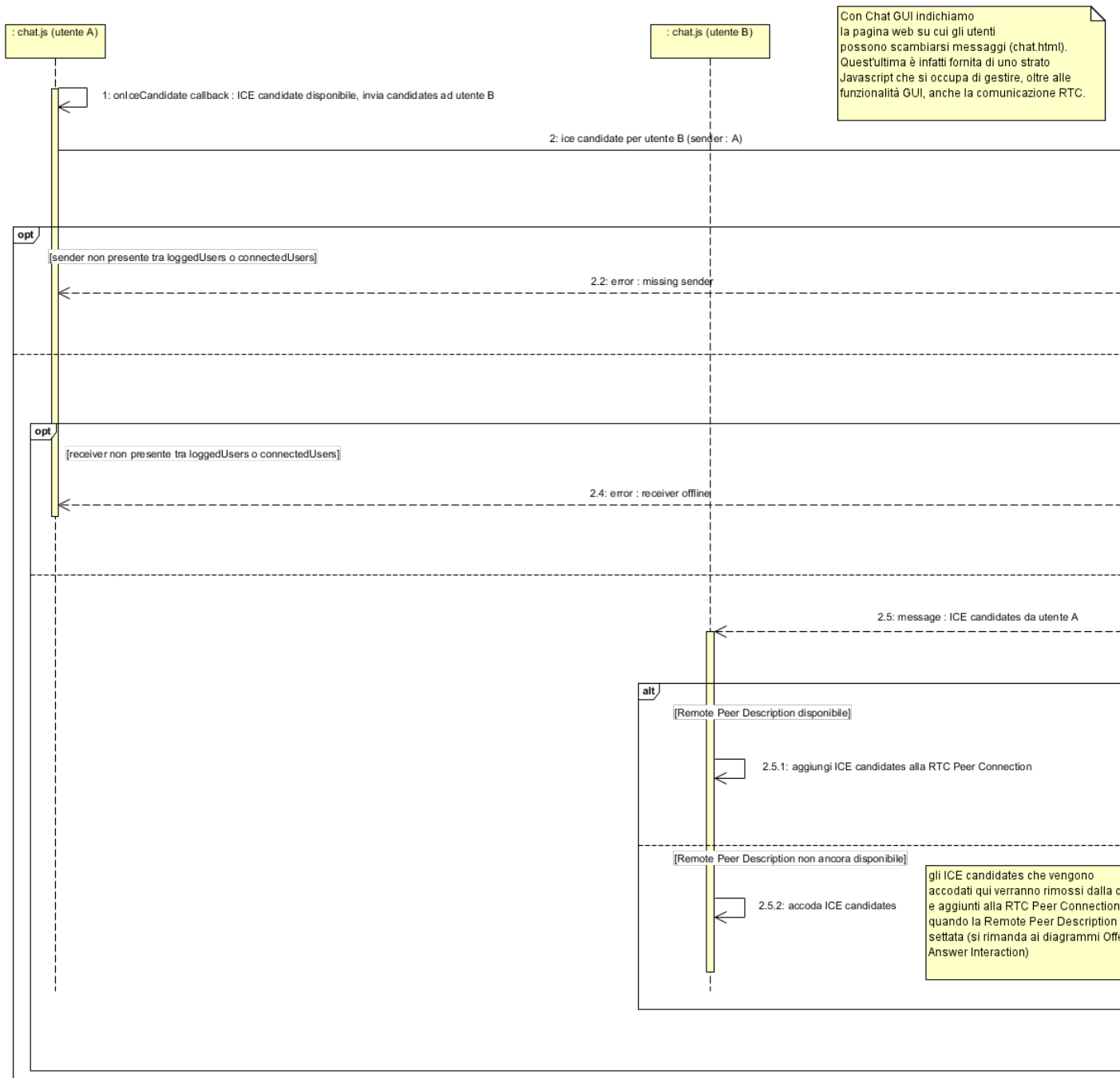


Figura 22 – Candidates Interaction

Entrambi i client posseggono dei *listener* per cui, non appena un ICE candidate risulta disponibile, viene inviato un messaggio verso il signaling server contenente informazioni a riguardo.

Il signaling server, dopo aver effettuato gli usuali controlli legati alla presenza degli utenti tra i *loggedUsers* e i *connectedUsers*, si occuperà semplicemente di inoltrare gli ICE candidates da un utente all'altro.

Tuttavia, visto che la configurazione degli ICE candidates può avvenire soltanto quando la *RemotePeerDescription* è stata settata, andremo ad accodare gli ICE candidates finché non otterremo una *RemotePeerDescription*: a quel punto preleveremo i candidates dalla coda e configureremo di conseguenza la nostra connessione verso l'altro peer.

Ulteriori dettagli possono essere ottenuti osservando la documentazione interna del server nodeJS.

2.2.1.4 Inactivity Timeout Interaction

Il sequence diagram riportato di seguito mostra il meccanismo con cui è possibile rimuovere automaticamente un utente inattivo se quest'ultimo non effettua operazioni per più di un certo periodo di tempo.

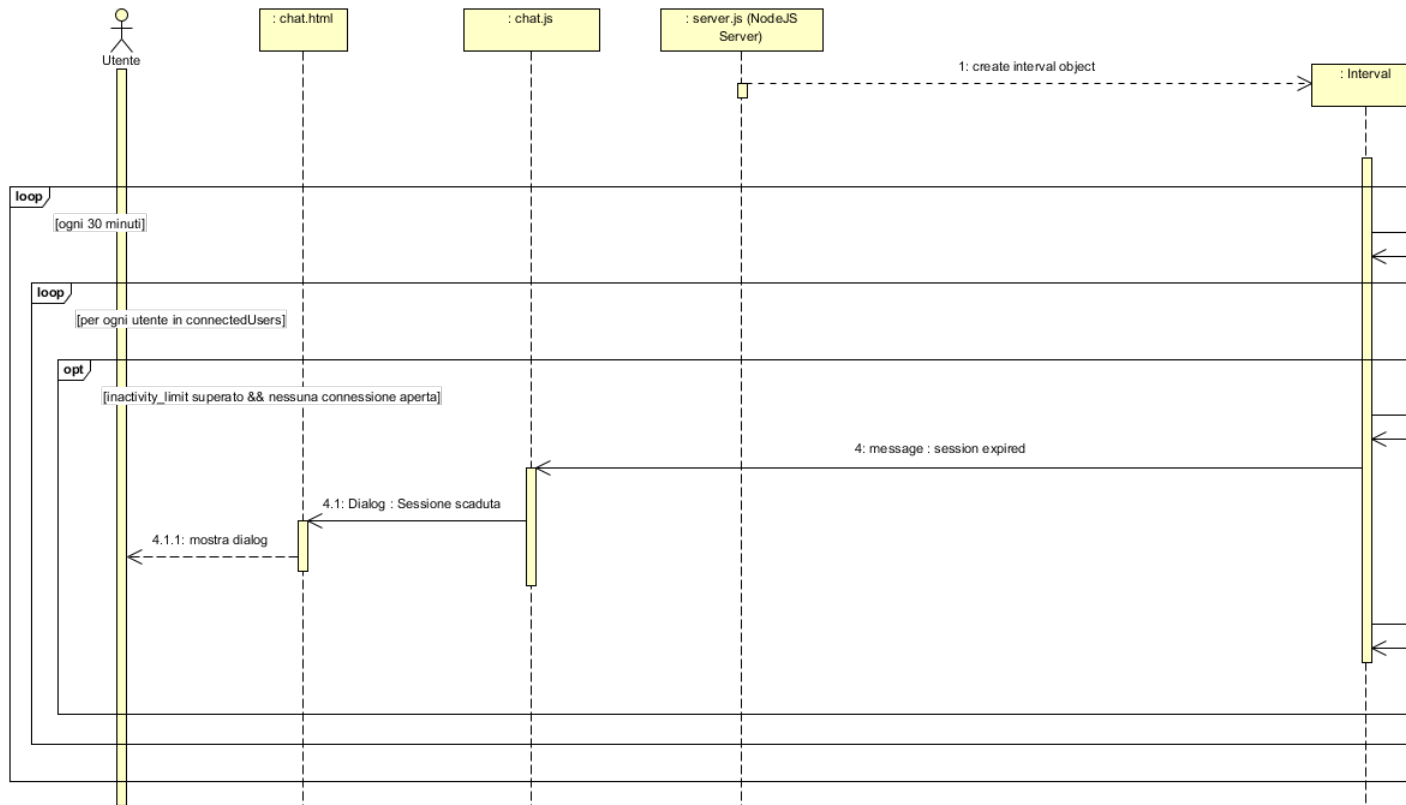


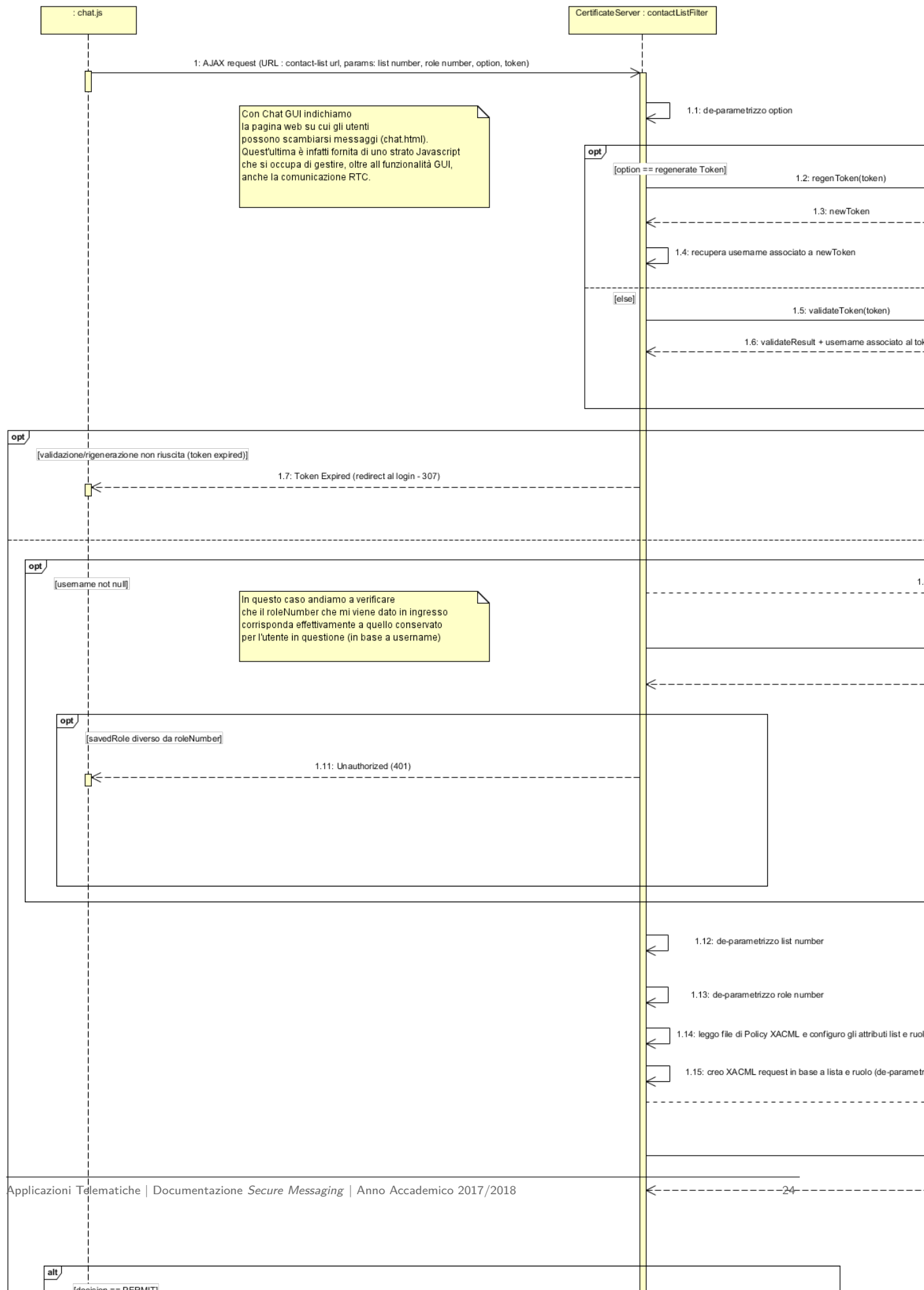
Figura 23 – Timeout Inactivity Interaction

Il timeout è stato programmato per scattare ogni 5 minuti.

Quando la callback associata al timeout viene invocata, andremo ad analizzare tutti gli utenti con connessioni attive verso il signaling server (presenti tra i `connectedUsers`) per identificare quegli utenti che sono rimasti *inattivi* (ovvero che non hanno inviato *answer/offer* e che non possiedono connessioni attive): tali utenti verranno quindi rimossi dai `loggedUsers` e dai `connectedUsers`, notificando l'evento con un apposito messaggio di stato lato client.

2.2.1.5 Contact List Filter

Il seguente diagramma mostra il comportamento del filtro `ContactListFilter`, utilizzato per l'accesso alle `contact-lists` che vengono prelevate dagli utenti ogni volta che si connettono alla chat, per poter selezionare (in base a policy ben precise) con quale utente comunicare.



Il filtro viene attivato ogniqualvolta viene richiesto l'accesso al contenuto della cartella `contact-lists`, in accordo con quanto dichiarato nel file `web.xml`.

Le richieste contengono tipicamente i seguenti parametri (POST):

- `list` : lista a cui accedere
- `role` : ruolo dell'utente richiedente
- `token`: token di accesso
- `option` : valore che indica rigenerazione/validazione del token

Alla ricezione della richiesta, verrà effettuata una de-parametrizzazione dei dati passati tramite POST (sfruttando una *Integer Access Reference Map*, accessibile [qui](#)).

Dopodichè, andremo a verificare se è stata richiesta la rigenerazione del token o la sua semplice validazione: nel caso in cui il token dovesse essere scaduto, entrambe queste procedure falliranno: questo produrrà il sollevamento di un'eccezione con cui signaleremo, lato client, un *temporary redirect* (307) che reindirizzerà l'utente al login.

Una volta assicuratici che il token di accesso è valido, andremo innanzitutto a verificare che il ruolo fornito dall'utente sia corrispondente al suo ruolo effettivo, dopodichè de-parametrizzeremo i dati che rappresentano lista e ruolo, per poi costruire la richiesta XACML che verrà sottoposta al PDP (*Policy Decision Point*), previa lettura del file `policy.xml` che contiene le XACML policies da applicare.

In caso di accesso consentito, il filtro si limita semplicemente ad allegare il token rinnovato (se presente) e a far avanzare correttamente la richiesta; in caso di permesso negato, invieremo comunque il token rinnovato (se presente), ma notificheremo il client con uno status code 401 (Unauthorized).

Per ulteriori dettagli, si rimanda alla documentazione interna del server nodeJS o del `contact list filter`.