POLITECNICO
MILANO 1863

Polo territoriale di Como

SCUOLA DI INGEGNERIA INDUSTRIALE E DELL'INFORMAZIONE

INGEGNERIA INFORMATICA

Corso di Ingegneria del Software

AuthOK

PROGETTO DEL CORSO DI INGEGNERIA DEL SOFTWARE

Parte III – Java

Ferrario Stefano

Gumus Tayfun

Isella Paolo

Martinese Federico

Indice

Introduzione

L'implementazione del codice Java di AuthOK è la parte finale del processo di realizzazione del software. Essa ha il compito di dar vita al programma reale che dovrà rispettare tutti i modelli precedentemente realizzati, a partire dal Goal Diagram fino all'ultimo dei diagrammi UML. Uno alla volta, tutti i metodi sono stati implementati e integrati tra di loro al fine di realizzare la logica applicativa del programma. L'implementazione prevede la creazione delle diverse classi suddivise nei package "Authorizer", "User", "Jsonrpc" e "Zeromq".

Aspetti particolari

- Il progetto è stato realizzato con l'IDE Intellij in un unico progetto suddiviso in 4 moduli: authorizer (server), user (client), jsonrpc (libreria di comunicazione), zeromq (libreria di canale). I moduli authorizer e user sono eseguibili tramite i metodi main rispettivamente delle classi Server e MainClass.
- Le impostazioni di test sul server si abilitano aggiungendo un argomento test all'avvio del programma. Quando in modalità test il server aggiunge alcune risorse di prove al suo elenco, riduce la durata dei token da 24 ore a 3 minuti, invoca la cancellazione periodica dei token ogni 3 minuti e stampa a video le operazioni che esegue. In modalità non test il server si avvia senza risorse disponibili, la durata dei token è di 24 ore, così come la periodicità della loro cancellazione e non viene stampato a schermo (sul server) il risultato delle operazioni, ma solo inviato al client
- La data di scadenza di una autorizzazione va specificata con il formato dd/MM/yy indicando solo due cifre per l'anno (che viene inteso essere del 21° secolo). L'ora di scadenza è la mezzanotte (00:00:00) del giorno indicato ed indica il momento da cui non è più possibile generare token con quella autorizzazione. Perciò un utente autorizzato fino al 25/05/18 non può richiedere token durante il giorno 25/05/18 in quanto la sua autorizzazione è scaduta dalla mezzanotte appena passata. I token richiesti il giorno precedente sono ancora validi fino alla loro naturale scadenza (24 ore in modalità standard)
- La modifica dell'id di una risorsa è un operazione che non produce risultato. Dato che la specifica jsonrpc definisce che una risposta, in caso di operazione a buon fine, deve avere un risultato, si è scelto di creare un risultato null. In caso di errori la risposta è definita normalmente con un oggetto errore.
- Da specifica l'utente non è abilitato a creare, modificare o cancellare risorse. Così come non può verificare la validità di un token.
- Gli utenti si identificano tramite il loro nome all'avvio del programma lato client. Nel caso sul server sia già presente un'autorizzazione assegnata a quel nome si assume che sia lo stesso.

- Da specifica è possibile che un utente chieda la creazione di autorizzazioni anche per altri utenti (specificandone il nome). Gli utenti possono quindi chiedere che gli sia inviata la chiave creata da qualcun altro tramite il comando verifica esistenza autorizzazione. Allo stesso modo possono verificare se la chiave è stata revocata
- I comandi per visualizzare lo stato dell'utente e del server non sono previsti dalle specifiche ma sono stati aggiunti per permettere di valutare più semplicemente il funzionamento del sistema.

Authorizer

Il modulo denominato authorizer è composto da:

- Package authorizer, che contiene una classe Server e un enum Methods
- Package gestoreRisorse, gestoreAutorizzazioni e gestoreRisorse

La struttura dei gestori è simile, in quanto tutti e tre permettono di creare, modificare, eliminare e verificare la validità e l' esistenza degli oggetti sotto il loro controllo.
Inoltre per mantenere la coerenza dei dati e della loro gestione le istanze dei gestori sono "accessibili" esclusivamente attraverso il metodo getIstance() che implementa il design pattern "singleton".

Server

La classe server è la classe principale del lato server del progetto.

Il costruttore del server ha due parametri: la porta da utilizzare per istanziare un server jsonrpc ed un parametro booleano per abilitare le impostazioni di test.

- Il metodo receive() riceve una lista di richieste dal server jsonrpc e, una alla volta, le esegue invocando il metodo selectMethod e ottenendo un risultato. Se la richiesta non è una notifica crea la risposta tramite il risultato ottenuto o, eventualmente, l'errore occorso e la aggiunge alla lista. Dopo aver eseguito tutte le richieste invia le risposte.
- Il metodo selectMethod(method, params) si occupa di smistare le richieste ai tre gestori ed ottenere i risultati. Controlla la validità dei parametri (numero e tipo) e l'esistenza del metodo richiesto
- Il metodo main viene lanciato all'avvio del programma lato server. Richiede una porta in input e ha un parametro opzionale "test" che abilita le impostazioni di test del server. Inoltre lancia un timer che cancella periodicamente i token scaduti.

Methods

Enum dei metodi invocabili sul server. Specifica la stringa con cui definire il metodo nelle richieste ed il numero di parametri necessari per invocarlo.

Gestore Autorizzazioni *[package]*

Il package *GestoreAutorizzazioni* contiene le classi *GestoreAutorizzazioni* e *Autorizzazione*. Include inoltre una classe *AuthorizationException* che estende la classe *Exception* per la gestione delle eccezioni.

Gestore autorizzazioni

La classe *GestoreAutorizzazioni* implementa tutti i metodi necessari per le operazioni che vengono eseguite sulle autorizzazioni. Il costruttore di questa classe utilizza il design pattern singleton per garantirne l'univocità. Durante l'istanziazione di un nuovo gestore viene creata una nuova HashMap autorizzazioni che conterrà le autorizzazioni create. I principali metodi implementati in questa classe sono:

- *genera_chiave_unica*

- *creaAutorizzazione*
- *revocaAutorizzazione*
- *verificaEisitenzaAutorizzazione*
- *verificaValiditàAutorizzazione*

Autorizzazione

La classe *Autorizzazione* ha al suo interno il costruttore necessario per la creazione di un nuovo token e i metodi per recuperarne le sue informazioni. Ogni autorizzazione è costituita da un nome utente, un livello di accesso e una scadenza e identificata nel gestore tramite chiave univoca.

Gestore Token *[package]*

Come nel precedente caso anche nel package *GestoreToken* sono presenti una classe "gestore" e una classe *Token*. Ancora una volta estendiamo la classe *Exception* per la generazione di *TokenExceptions*

Gestore token

La classe *GestoreToken* implementa tutti i metodi necessari per le operazioni che vengono eseguite sui token. Il costruttore di questa classe utilizza il design pattern singleton per garantirne l'univocità. Durante l'istanziazione di un nuovo gestore viene creata una nuova HashMap che conterrà i token creati.
I metodi implementati in questa classe sono:

- *creaToken*
- *verificaToken*
- *cancellaTokenScaduti*
- *cancellaTokenChiave*

Token

La classe *Token* ha al suo interno il costruttore necessario per la creazione di un nuovo token e i metodi per recuperarne le sue informazioni. Ogni token è costituito dalla chiave dell'autorizzazione corrispondente, dalla risorsa per cui è stato creato e dalla data e ora di creazione. Viene identificato nel gestore tramite un identificativo univoco.

Gestore Risorse *[package]*

Per la creazione delle risorse si ricorre a un design pattern factory method. Pertanto per ogni tipo di risorsa vi sarà una classe "risorsa" e una classe "factory" dedicata. Per facilitare la gestione dei tipi di risorsa disponibili abbiamo introdotto un Enum che li identifica. Come nel caso precedente sono presenti anche qui una classe "gestore", contenente i diversi metodi per la creazione e per le operazioni sulle risorse, e una classe *ResourceException* che estende la classe *Exception*.

Gestore risorse

La classe *GestoreRisorse* presenta un costruttore di tipo singleton che all'istanziazione crea un'HashMap *databaseRisorse* la quale conterrà i dati relative alle risorse create. Il *GestoreRisorse* si occupa della creazione, modifica e cancellazione di una risorsa e della verifica dell'esistenza di una risorsa dato l'ID.

Risorsa [abstract]

La classe *Risorsa* è di tipo abstract e viene estesa, nel nostro caso specifico dalle classi *RisorsaFibonacci*, *RisorsaLanciaDado*, *RisorsaLink*. Oltre al costruttore tale classe contiene anche i metodi necessari per recuperare e modificare le informazioni che costituiscono una risorsa ovvero il livello, l'ID.

Risorse concrete

I tipi di risorsa concreta non sono specificati dalle specifiche e ne sono stati implementati alcuni esempi ai fini di test.

Jsonrpc

Il modulo jsonrpc contiene il package jsonrpc e un package di test Junit. Utilizza una libreria json. Nel package j*sonrpc* viene implementata la libreria necessaria alla comunicazione tra client e server usando in maniera completa le specifiche di jsonrpc. È composto dalle classi Server e Client che realizzano la comunicazione e dalle classi usate per definire i messaggi che vengono inviati. Inoltre è presente una JSONRPCException.

Server

Implementa l'interfaccia Iserver.
Il costruttore ha un parametro intero porta che viene utilizzato per istanziare un ZmqServer. Nel caso in cui la porta sia occupata viene gestita l'eccezione.
Il metodo receive() restituisce le richieste ricevute. Ottiene la stringa ricevuta dal ZmqServer e la analizza:

- JSONArray: crea un oggetto batch e restituisce le richieste del batch come ArrayList<Request>
- JSONObject: crea una singola richiesta, la inserisce in una lista e restituisce la lista
- Nel caso si verifichino errori di parsing crea una risposta con errore PARSE, la invia tramite il ZmqServer e restituisce una lista di richiesta valide vuota

Esistono due metodi reply() in overload, uno invia una risposta singola e l'altro invia una lista di risposte tramite batch.
Entrambi verificano che la o le risposte siano compatibili con l'ultima ricezione effettuata dal server: non è possibile inviare un numero di risposte diverso dal numero di richieste non notifiche ricevute. In caso contrario viene lanciata un'eccezione di tipo JSONRPCException.

Client

Implementa l'interfaccia IClient.
Il costruttore ha un parametro intero porta che viene utilizzato per istanziare un ZmqClient.
Il metodo sendNotify(Request notify) invia una notifica tramite il client zmq, senza restituire risultati. La richiesta deve necessariamente essere notifica.
Il metodo sendRequest(Request req) invia una richiesta e restituisce una risposta. La richiesta non può essere notifica. Se la stringa ricevuta dal client zmq dopo l'invio è nulla significa che il server non è raggiungibile, altrimenti ricostruisce la risposta partendo dalla stringa ricevuta, gestendo gli errori di parsing.
Il metodo sendBatch(ArrayList<Request> reqs) permette l'invio di richieste multiple e la ricezione delle risposte corrispondenti. Crea un oggetto batch partendo dalla lista di richieste e lo invia sul client zmq. Restituisce la lista delle risposte. Come il metodo sendRequest() gestisce errori di parsing e server irraggiungibile, inoltre è gestito il caso di una lista di sole notifiche che non produce risposte.

JsonRpcObj

Classe astratta che modella un generico oggetto jsonrpc. Definisce gli attributi jsonobj e jsonstring e i rispettivi getter. Definisce un metodo astratto toJsonObj().
Implementa dei metodi statici usati nella gestione degli oggetti jsonrpc:

- checkMembersSubset() verifica che un dato oggetto json non contenga membri non previsti

- putMember e putStructuredMember inseriscono in un dato array o object un membro o un membro strutturato

JsonRpcMessage

Classe astratta che estende JsonRpcObj. Descrive un generico messaggio jsonrpc, ossia un jsonrpcobj dotato di id. Specifica la versione di jsonrpc (2.0).

AbstractRequest e AbstractResponse

Classi astratte che estendono JsonRpcMessage e descrivono, rispettivamente, una richiesta ed una risposta.
Contengono enum dei membri di una richiesta (“id”, “method”, “jsonrpc”, “params”) e di una risposta (“id”, “result”, “error”, “jsonrpc”).
Definiscono un costruttore tramite parametri che li imposta come attributi, i relativi getter ed il metodo equals(). Una richiesta con id null è una notifica.

Request e Response

Classi che estendono rispettivamente AbstractRequest e AbstractResponse. Implementano un costruttore con un parametro stringa che ricostruisce la richiesta/risposta a partire da una stringa json, valutandone la correttezza. Implementano il metodo astratto toJsonObj() definito nella classe JsonRpcObj.

Error

Estende la classe JsonRpcObj. Rappresenta un oggetto jsonrpc errore utilizzato dalla classe Response.
Definisce gli enum ErrMembers, che specifica i membri di un oggetto errore (“code”, “message”, “data”); e Errors, che specifica gli errori predefiniti (parse, invalid request, method not found, invalid params, internal error).
Ha costruttori che impostano i parametri code, message e data (quest’ultimo opzionale) ed uno che li ricava a partire da un oggetto json.
Implementa il metodo toJsonObj() definito nella classe JsonRpcObj().

Id

Rappresenta l’id di un messaggio jsonrpc.
Può essere intero, stringa o di tipo nullo. Il tipo nullo non identifica una notifica: è usato nelle risposte quando non si è in grado di recuperare l’id della richiesta. Ad esempio ad una richiesta con json invalido, da cui non si riesce a risalire all’id, si risponde con una risposta con id nullo ed errore invalid request.
Il tipo di Id è indicato da un enum apposito.
Il metodo statico getIdFromRequest(String request) cerca un id valido nella stringa passata, anche se questa dovesse essere una richiesta invalida. Nel caso non sia possibile restituisce un id di tipo nullo.

StructuredMemeber

Rappresenta un membro strutturato della specifica jsonrpc (in particolare usato per i parametri della richiesta). È o array o mappa e può essere costruito sia con una ArrayList/HasMap sia con un JSONArray/JSONObject.
Il metodo statico toStructuredMember costruisce uno StructuredMember a partire da un oggetto di tipo Object.

Member

Membro generico di un oggetto jsonrpc. Può essere primitivo (null, stringa, numerico, booleano) o strutturato (StructuredMember array o mappa). Il tipo è indicato dall'enum Types.

I getter del valore lanciano ClassCastException se il membro non è del tipo previsto.

Ilmetodo statico toMember converte un oggetto di tipo Object in un Member, se di tipo idoneo.

Batch

Gestisce le comunicazioni di tipo batch: array di richieste inviato dal client a cui il server risponde con un array di risposte.
Viene costruito a partire da un JSONArray o ArrayList di richieste, con cui popola la lista di richieste privata. Nel caso in cui una richiesta non sia valida viene aggiunta una risposta con errore invalid request nella lista delle risposte nella posizione corrispondente.
Il metodo put inserisce tutte le risposte mancanti nella lista delle risposte, saltando quelle corrispondenti alle notifiche. Le risposte devono essere del numero esatto (num richieste – num richieste invalide – num notifiche).

JsonRpcException

Estende la classe Exception, eccezione riguardante la specifica jsonrpc.

Zeromq

Il modulo zeromq comprende l'omonimo package e si occupa della gestione del canale di comunicazione.
All'interno del package si sviluppano due classi principali: la classe *ZmqClient* e la classe *ZmqServer*. Esse implementano rispettivamente le interfacce *IZmqClient* e *IZmqServer*.
Per la creazione di entrambe le classi si fa uso della libreria *org.zeromq.ZMQ (jeromq)*.

ZmqServer

Il costruttore per istanziare un nuovo ZmqServer richiede un intero che identifica la porta su cui il server dovrà ascoltare. Per farlo crea un Zmq.Context ed un Zmq.Socket di tipo ROUTER ed esegue l'operazione di binding.
Il metodo receive() riceve il prossimo messaggio Zmsg ne estrae il frame identity, l'eventuale frame vuoto ed il frame di messaggio. Memorizza i primi due, rendendo possibile rispondere in un secondo momento, e restituisce il terzo convertendolo a stringa.
Il metodo reply(String string) risponde all'ultima richiesta ricevuta. Per costruire il messaggio utilizza il frame identity e l'eventuale frame vuoto che erano stati salvati durante la ricezione. A questi aggiunge un nuovo messaggio costruito con la stringa da inviare. Infine resetta i frame identity e vuoto per impedire di inviare più risposte ad una sola richiesta. Il metodo può lanciare una UnsupportedOperationException nel caso in cui non ci sia un frame identity memorizzato, ossia quando si tenta di eseguire le operazioni di ricevi e rispondi in un ordine non valido (ogni risposta deve seguire una ricezione)

ZmqClient

Per l'istanziazione di un nuovo *ZmqClient* sarà necessario un costruttore che, come nel caso precedente riceve in ingresso un intero che definisce la porta.
Il metodo request(String req) definisce un socket di tipo REQ e lo connette all'indirizzo localhost su porta specificata. Invia la richiesta, imposta un tempo di attesa massimo per ricevere una risposta e la restituisce. Nel caso in cui il tempo scada prima di ricevere la risposta, restituisce un valore null.
Il metodo send(String msg) è simile al metodo request ma definisce un socket di tipo DEALER e invia senza non ricevere risposta.

User

Il package user si occupa della gestione del lato client del sistema.

MainClass

Questa classe si occupa dell'interazione con l'utente. Contiene il main del programma che stampa un menu di comandi ed i risultati delle loro invocazioni. All'acquisizione di un nome, viene creato un nuovo utente e viene stampato un elenco di operazioni a ciascuna delle quali è associato un numero. La verifica dell'input è effettuata tramite espressioni regolari.
I metodi della *MainClass* per soddisfare le richieste dell'utente interagisce con un oggetto di tipo *CreatoreRichieste* tramite l'interfaccia IntUtente.

CreatoreRichieste

Il *CreatoreRichieste* definisce i metodi che, a fronte di un comando dell'utente, elaborano una nuova richiesta da inviare al server. In questa classe sono presenti i metodi per ogni operazione che può essere eseguita. Utilizza un client jsonrpc per inviare le richieste e ricevere le risposte. Questa classe implementa le due interfacce *IUtente* e *IAdmin*. La seconda non viene utilizzata dal sistema in quanto da specifiche non viene richiesto di implementare un'utente in grado di eseguire i comandi di creazione, modifica e cancellazione delle risorse.

Utente

La classe *Utente* contiene il costruttore per la creazione di un nuovo utente, caratterizzato da un nome utente che supponiamo univoco. Ogni utente ha inoltre un parametro *chiave* e un'HashMap *tokens* per salvare rispettivamente la chiave e i token posseduti.

Anche in questo caso ricorriamo ad una classe *AuthorizerException* che ci permette di estendere la classe predefinita *Exception* per la gestione delle eccezioni nel package *User*