

*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 3](#_Toc501643765)

[Authorizer 4](#_Toc501643766)

[Gestore Autorizzazioni *[package]* 4](#_Toc501643767)

[**Gestore autorizzazioni** 4](#_Toc501643768)

[**Autorizzazione** 4](#_Toc501643769)

[Gestore Token *[package]* 4](#_Toc501643770)

[**Gestore token** 4](#_Toc501643771)

[**Token** 5](#_Toc501643772)

[Gestore Risorse *[package]* 5](#_Toc501643773)

[**Gestore risorse** 5](#_Toc501643774)

[**Risorsa** [abstract] 5](#_Toc501643775)

[MethodUtils 5](#_Toc501643776)

[Server 5](#_Toc501643777)

[Jsonrpc 6](#_Toc501643778)

[Server 6](#_Toc501643779)

[Client 6](#_Toc501643780)

[JsonRpcObj 6](#_Toc501643781)

[JsonRpcMessage 6](#_Toc501643782)

[AbstractRequest 6](#_Toc501643783)

[AbstractResponse 6](#_Toc501643784)

[Id 6](#_Toc501643785)

[StructuredMemeber 7](#_Toc501643786)

[Batch 7](#_Toc501643787)

[Error 7](#_Toc501643788)

[JsonRpcException 7](#_Toc501643789)

[Zeromq 8](#_Toc501643790)

[ZmqServer 8](#_Toc501643791)

[ZmqServer 8](#_Toc501643792)

[User 9](#_Toc501643793)

[Class 9](#_Toc501643794)

**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”.

**Authorizer**

Il package denominato **authorizer** è composto da:

* Classi che rappresentano i **token**, le **autorizzazioni** e le **risorse**
* Classi per la gestione degli oggetti sopracitati
* Una classe **server**

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”.

La classe **server** ha (tra gli altri) il compito di invocare i metodi messi a disposizione dai gestori e comunicare i risultati al “lato client” del programma.

## **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 nuove autorizzazioni create per consentirne il salvataggio e le successive verifiche su di essi. 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.

## **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 per consentirne il salvataggio e le successive verifiche su di essi. 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.

## **Gestore Risorse** *[package]*

Il fulcro di questo package sono ovviamente le **risorse.** La creazione di istanze di risorse è stata progettata implementando il design pattern “factory method”. La creazione di istanze delle varie specializzazione della classe **abstract risorsa** viene così standardizzata e permette in futuro di aggiungere nuovi tipi di risorse senza apportare sostanziali modifiche al codice presente.

Un secondo aspetto da sottolineare è l’ utilizzo della struttura dati “hashmap” che contiene tutte le istanze delle risorse gestite dal programma. Abbiamo deciso di utilizzare questa struttura dati perché creare associazioni tra le risorse e gli ID che le identificano (univocamente) rende la gestione degli oggetti molto intuitiva e soprattutto poco dispendiosa dal punto di vista delle righe di codice scritte (il pezzo sulla hash può essere esteso a tutte le hashmap dei vari gestori quindi si potrebbe anche spostare nell’ introduzione dell’ authorizer, vedi tu)

## **Method**

Abbiamo scelto di ricorrere ad una classe *Method* per definire una *ENUM* con i diversi metodi che il server può invocare. Nella stesa classe definiamo il formato data (*DATE\_FORMAT*), il formato date e ora (*DATE\_HOUR\_FORMAT*) e la porta del server (*PORT*).

## **Server**

La classe *Server* presenta un costruttore di tipo singleton che ne garantisce l’univocità. All’interno di questa classe troviamo i metodi *receive*, *selectMethod* e *getServerStatus*.

Per facilitare le fasi di test dell’applicativo è presente il metodo *setTest*, quindi un metodo *isTest* che ne controlla lo stato.

All’accensione del Server esso si mette in ascolto di nuove richieste. È presente nel *main* di queste classe anche un *Timer* che esegue una scansione periodica ed elimina i token scaduti.

**Jsonrpc**

Nel package *Jsonrpc* vengono implementate le classi necessarie alla comunicazione tra client e server. Qui possiamo notare una classe *Server* ed una classe *Client* che implementano le rispettive interfacce *IServer* e *IClient*. È strata creata una classe *JsonRpcObj* che viene poi estesa dalla classe *JsonRpcMessage*. Per la gestione di richieste e risposte vi sono le classi astratte *AbstractRequest* e *AbstractResponse* che vengono estese, per consentirne l’utilizzo, nelle classi *Request* e *Response*

## **Server**

Per l’istanziazione di un nuovo *ZmqServer* sarà necessario un costruttore che riceve in ingresso un intero che definisce la porta. La classe *ZmqServer* implementa i metodi *receive* che riceve i messaggi in arrivo e una funzione *reply* che consente l’invio di risposte. Affinchè il sistema possa ricevere più più di una richiesta prima di inviare una nuova risposta introduciamo un valore *identity*.

## **Client**

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. In questa classe troviamo poi i metodi *request* e *send* per l’invio di richieste e notifiche rispettivamente.

## **JsonRpcObj**

Questa classe astratta permette la gestione degli oggetti JSON e la generazione della stringa JSON che viene utilizzata durante la comunicazione.

## **JsonRpcMessage**

Estende la classe precedentemente descritta… [Descrizione]

## **AbstractRequest**

[Descrizione]

## **AbstractResponse**

[Descrizione]

## **Id**

[Descrizione]

## **StructuredMemeber**

[Descrizione]

## **Batch**

[Descrizione]

## **Error**

[Descrizione]

## **JsonRpcException**

[Descrizione]

**Zeromq**

All’interno del package *zeromq*  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*.

## **ZmqServer**

Per l’istanziazione di un nuovo *ZmqServer* sarà necessario un costruttore che riceve in ingresso un intero che definisce la porta. La classe *ZmqServer* implementa i metodi *receive* che riceve i messaggi in arrivo e una funzione *reply* che consente l’invio di risposte. Affinchè il sistema possa ricevere più più di una richiesta prima di inviare una nuova risposta introduciamo un valore *identity*.

## **ZmqServer**

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. In questa classe troviamo poi i metodi *request* e *send* per l’invio di richieste e notifiche rispettivamente.

**User**

## **MainClass**

In questa classe è presente uno scanner che riceve i comandi dell’utente. 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. L’utente per selezionare l’operazione desiderata dovrà inserire il numero corrispondente. I metodi della *MainClass* per soddisfare le richieste dell’utente interagisce con un oggetto di tipo *CreatoreRichieste*.

## **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. 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 Admin.

## **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*

**Identificativo gruppo**: 10

**Progetto**: Tema A

**Componenti**:

845386 Ferrario Stefano [stefano6.ferrario@mail.polimi.it](mailto:stefano6.ferrario@mail.polimi.it)

843994 Gumus Tayfun [tayfun.gumus@mail.polimi.it](mailto:tayfun.gumus@mail.polimi.it)

854412 Isella Paolo [paolo.isella@mail.polimi.it](mailto:paolo.isella@mail.polimi.it)

845326 Martinese Federico [federico.martinese@mail.polimi.it](mailto:federico.martinese@mail.polimi.it)