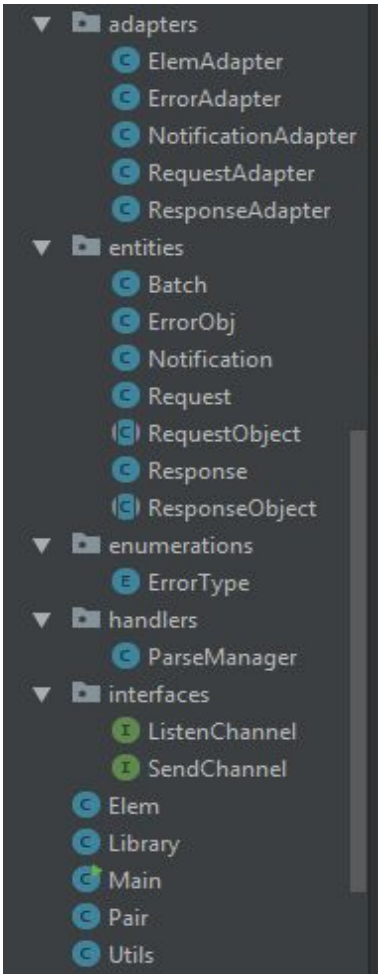
Anno accademico 2017-18
Prova finale di Ingegneria del software
Prof. Marco Brambilla, Como
#swEngComo

Gruppo 4
Tema A

DOCUMENTAZIONE DEL PROGETTO JAVA

L'intero progetto è diviso in due sotto progetti: uno dei due contiene il codice sorgente della Libreria - *Library* - il quale viene integrato opportunamente nel codice sorgente dell'Autenticatore - *Authenticator* - . La Libreria, come spiegato in seguito, è stata progettata completamente agnostica al tipo di progetto a cui viene integrata, oltre che, come da specifiche, al canale di comunicazione che si desidera implementare, e cerca di gestire tutti i possibili casi ed errori derivabili dall'utilizzo del protocollo JSON-RPC.

TEMA COMUNE: Libreria - *Library*



Verrà spiegato in primis il comportamento della Libreria nei confronti di un qualsiasi progetto fruitore della stessa, successivamente il comportamento interno delle classi e dei rispettivi metodi.

Interfacce

Library presenta due interfacce che finalizzano la specifica di comunicazione remota fra due sistemi: *ListenChannel* e *SendChannel.*

```
public interface ListenChannel {

    void bind(String ip, short port);
    group_4.Pair<String, String> receive();
    void reply(String identity,String message);
    void close();

}
```

```
public interface SendChannel {

    void connect(String ip, short port);
    String getChannelIdentity();
    void send(String message);
    String receive();
    void close();

}
```

Completamente agnostiche al canale di comunicazione le interfacce espongono i metodi di connessione, ascolto, ricezione, invio e risposta che il progetto, che integrerà la Libreria, dovrà implementare scegliendo l'opportuna piattaforma di comunicazione. L'interfaccia *ListenChannel* verrà implementata dal lato Server, viceversa, la *SendChannel* dal lato Client.
Il metodo *receive()* torna la stringa Json-rpc (e l'identity del Client nel caso del metodo lato Server); allo stesso modo ai metodi *send()* e *reply()* verrà passata la stringa json-rpc da inviare (e l'identity del Client a cui il Server deve rispondere).

Library.class

La libreria presenta una classe Facade -*Library*- che espone tutti i metodi implementati nella libreria stessa e utilizzabili dal progetto integratore:

```
public class Library {

    public static Request parseRequest(String json) { return ParseManager.getInstance().parseRequest(json); }
    public static Response parseResponse(String json) { return ParseManager.getInstance().parseResponse(json); }
    public static ErrorObj parseError(String json) {return ParseManager.getInstance().parseErrorObj(json);}
    public static String reqToJson(Request request) {return ParseManager.getInstance().requestToJson(request);}
    public static String responseToJson (Response response) {return ParseManager.getInstance().responseToJson(response);}
    public static String errorToJson (ErrorObj errorObj) {return ParseManager.getInstance().errorToJson(errorObj);}
    public static String batchToJson (Batch batch) {return ParseManager.getInstance().batchToJson(batch);}
    public static Batch parseBatch (String json) {return ParseManager.getInstance().parseBatch(json);}
    public static Notification parseNotification (String json) {return ParseManager.getInstance().parseNotification(json);}
    public static String notificationToJson (Notification notification) {return  ParseManager.getInstance().notificationToJson(notification);}

}
```

La Facade espone i metodi di parsing necessari a trasformare una stringa json-rpc negli oggetti messaggio definiti nella documentazione del protocollo JSON-RPC 2.0: *Request, Notification, Response, ErrorObj, Batch.* Complementariamente espone i metodi "**ToJson()*" che permettono la conversione di un qualunque oggetto messaggio in stringa json-rpc. Il sistema fruitore della libreria può quindi delegare tutti i problemi di parsing e reflection del Json-rpc alla libreria stessa.

Internamente la libreria sfrutta quindi un *ParseManager* (classe Singleton) che usufruisce di una libreria esterna realizzata da Google per la serializzazione/deserializzazione di oggetti JAVA in/da JSON: Gson.

```
import com.google.gson.*;
```

Uno qualsiasi dei metodi esposti da *Library.class* si presenta in questo modo:

SERIALIZZAZIONE

```
public Request parseRequest(String json) {
    Gson gson = Utils.getGson();
    try {
        Request request = gson.fromJson(json, Request.class);
        return request;
    } catch (JsonParseException e){
        System.out.println(e.getMessage());
        return null;
    }
}
```

DESERIALIZZAZIONE

```
public String requestToJson(Request request){
    Gson gson = Utils.getGson();
    try {
        String json = gson.toJson(request, Request.class);
        return json;
    } catch (JsonParseException e) {
        System.out.println(e.getMessage());
        return null;
    }
}
```

Il punto cruciale e di maggiore difficoltà affrontato durante la progettazione della libreria è stato adattare la serializzazione/deserializzazione di Gson nel formato JSON-RPC 2.0 (Gson serializza/deserializza nel formato JSON standard), seguendo le regole del protocollo e gestendo tutti i casi di errore di formato.

Adapters

Il metodo statico *getGson()* contenuto nella classe *Utils* non torna un oggetto Gson standard:

```
public static Gson getGson() {
    GsonBuilder gsonBuilder = new GsonBuilder();
    gsonBuilder.registerTypeAdapter(Elem.class, new ElemAdapter());
    gsonBuilder.registerTypeAdapter(Request.class, new RequestAdapter());
    gsonBuilder.registerTypeAdapter(ErrorObj.class, new ErrorAdapter());
    gsonBuilder.registerTypeAdapter(Response.class, new ResponseAdapter());
    gsonBuilder.registerTypeAdapter(Notification.class, new NotificationAdapter());
    return gsonBuilder.create();
}
```

Gson dà la possibilità di registrare degli adapters, implementabili direttamente da chi utilizza la libreria; questi permettono una serializzazione/deserializzazione “custom” di oggetti Java progettabile direttamente dal programmatore, eventualmente potendo gestire tutti gli errori di formato.
Oltre che gli adapters per i tipi di oggetto messaggio JSON-RPC - creati semplicemente per semplificare la gestione degli errori- abbiamo progettato uno specifico adapters per i “params”/”result” - ovvero il campo dei parametri/risultati di una stringa json-rpc- che è l’unico che analizzeremo in seguito.

ElemAdapters

Secondo il protocollo JSON-RPC 2.0 il campo params/result deve un “valore strutturato”; semplificando l’oggetto parametro che ne definisce il campo json-rpc può contenere:
-un oggetto primitivo di tipo String, Number o booleano
-un arrayList di oggetti primitivi o di oggetti parametri
-un mappa la cui chiave è una stringa e il cui valore è un oggetto primitivo o un oggetto parametro.
L’oggetto parametro implementato da noi è il seguente:

```java
public class Elem {

    private Object primitive = null;
    private ArrayList<Elem> elemArrayList = null;
    private Map<String, Elem> elemMap = null;

    public Elem(){

    }

    public Elem(String primitive) {
        this.primitive = primitive;
        elemArrayList = null;
        elemMap = null;
    }

    public Elem(Number primitive) {
        this.primitive = primitive;
        elemArrayList = null;
        elemMap = null;
    }

    public Elem(Boolean primitive) {
        this.primitive = primitive;
        elemArrayList = null;
        elemMap = null;
    }
```

```java
public Elem(Elem elem) { add(elem); }

public Elem(String key, Elem elem) { add(key, elem); }

public void add(Elem element) {
    if (elemArrayList==null) {
        elemArrayList = new ArrayList<Elem>();
        elemMap = null;
        primitive = null;
    }

    elemArrayList.add(element);
}

public void add(String key, Elem element) {
    if (elemMap==null) {
        elemMap = new HashMap<String, Elem>();
        elemArrayList = null;
        primitive = null;
    }

    elemMap.put(key, element);
}
```

Il relativo adapter progettato è interessante poiché incorpora un algoritmo ricorsivo che richiama se stesso ogni volta che il parametro non è primitivo (caso base):

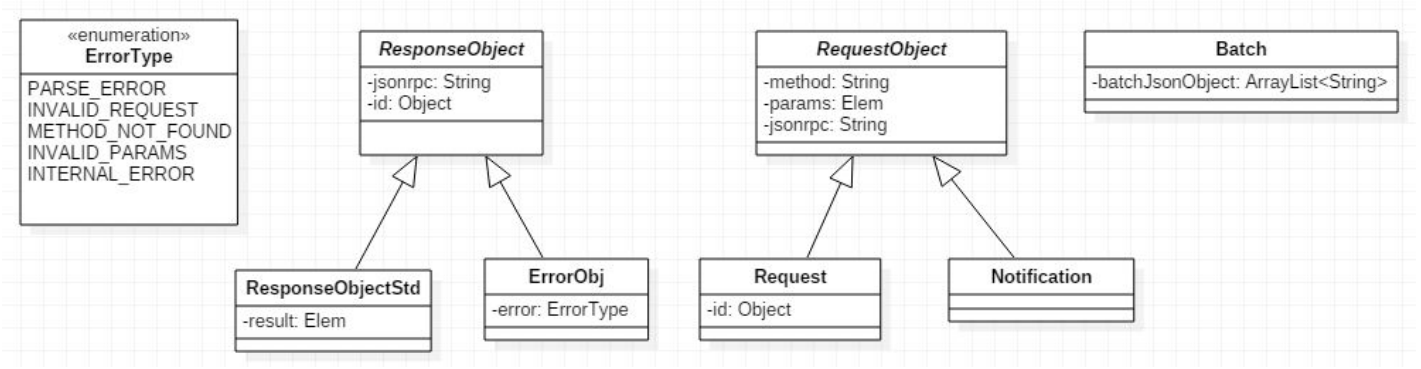
```java
    public Elem deserialize(JsonElement jsonElement, Type type, JsonDeserializationContext jsonDeserializationContext) throws JsonParseException {
        return parseElem(jsonElement, jsonDeserializationContext);
    }

    public JsonElement serialize(Elem elem, Type type, JsonSerializationContext jsonSerializationContext) {
        return parseJson(elem, jsonSerializationContext);
    }
}
```

```
public class ElemAdapter implements JsonDeserializer<Elem>, JsonSerializer<Elem> {

    private Elem parseElem(JsonElement jsonElement, JsonDeserializationContext jsonDeserializationContext) {

        if (jsonElement.isJsonPrimitive()) {
            JsonPrimitive jsonPrimitive = jsonElement.getAsJsonPrimitive();
            if (jsonPrimitive.isNumber()) {
                Number number = jsonDeserializationContext.deserialize(jsonElement, Number.class);
                return new Elem(number);
            } else if (jsonPrimitive.isString()) {
                String string = jsonDeserializationContext.deserialize(jsonElement, String.class);
                return new Elem(string);
            } else if (jsonPrimitive.isBoolean()) {
                Boolean b = jsonDeserializationContext.deserialize(jsonElement, Boolean.class);
                return new Elem(b);
            }
        } else if (jsonElement.isJsonObject()) {
            Elem result = null;
            JsonObject jsonObject = jsonElement.getAsJsonObject();
            Set<String> keys = jsonObject.keySet();
            for (String key : keys) {
                JsonElement element = jsonObject.get(key);
                Elem elem = parseElem(element, jsonDeserializationContext);
                if (result==null) {
                    result = new Elem(key, elem);
                } else {
                    result.add(key, elem);
                }
            }
            return result;
        } else if (jsonElement.isJsonArray()) {
            JsonArray jsonArray = jsonElement.getAsJsonArray();
            Elem result = null;
            for (JsonElement element : jsonArray) {
                Elem elem = parseElem(element, jsonDeserializationContext);
                if (result==null) {
                    result = new Elem(elem);
                } else {
                    result.add(elem);
                }
            }
            return result;
        }
        return null;
    }
```

```
private JsonElement parseJson(Elem elem, JsonSerializationContext jsonSerializationContext){

    if(elem.isMap()){
        JsonObject jsonObject = new JsonObject();
        Set<String> keys = elem.getSetKey();
        for (String key : keys){
            JsonElement element = parseJson(elem.getElemByKey(key),jsonSerializationContext);
            jsonObject.add(key,element);
        }
        return jsonObject;
    } else if (elem.isArray()){
        JsonArray jsonArray = new JsonArray();
        for (int i = 0; i < elem.size(); i++){
            Elem eleml = elem.getElemAt(i);
            if (eleml != null) {
                JsonElement jsonElement = parseJson(eleml, jsonSerializationContext);
                jsonArray.add(jsonElement);
            }
        }
        return jsonArray;
    } else if (elem.getNumber() != null){
        Number number = elem.getNumber();
        return jsonSerializationContext.serialize(number, Number.class);
    } else if (elem.getString() != null){
        return jsonSerializationContext.serialize(elem.getString(), String.class);
    } else if (elem.getBoolean() != null){
        return jsonSerializationContext.serialize(elem.getBoolean(), Boolean.class);
    }
    return null;
}
```

Entities package

All'interno di questo package sono contenuti i modelli degli oggetti messaggio JSON-RPC implementati come specificato nella documentazione UML:

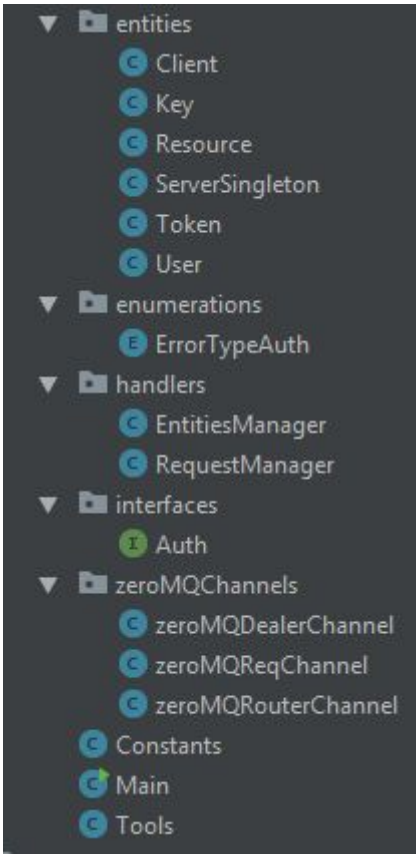


Guida all'utente

Una volta importata la libreria all'interno di un progetto lo sviluppatore fruitore della stessa deve assicurarsi di implementare le interfacce di comunicazione remota qui sopra spiegate; la libreria lascia libero arbitrio alla scelta del canale di comunicazione. Inoltre la libreria fornisce gli strumenti

necessari sia per la gestione di tutti gli errori di formato json-rpc, sia per quella degli errori interni al progetto fruitore (vedi Gestione errori doc. TEMA A).

TEMA A: Autenticatore - *Authenticator*



Comunicazione remota

```
import org.zeromq.ZMQ;
```

Il canale di comunicazione implementato per la comunicazione Client/Server è ZeroMQ, come suggerito nelle specifiche; il package *zeroMQChannels* contiene le classi implementatrici delle due interfacce di comunicazione della *Library: ListenChannel* e *SendChannel* (vedi sopra).
I socket dei canali sfruttati dal lato Client possono essere di tipo REQ o DEALER:

```
public class zeroMQReqChannel implements SendChannel
```

```
public class zeroMQDealerChannel implements SendChannel
```

Il primo elabora l'invio e la ricezione di messaggi in lockstep, viceversa il secondo può inviare e ricevere messaggi in maniera asincrona (http://zeromq.org/intro:read-the-manual : ZMQ doc).
N.B. L'autenticatore progettato utilizzerà lato Client esclusivamente *zeroMQReqChannel* poiché il Client non ha bisogno di una comunicazione asincrona con il Server: inviata una richiesta aspetterà sempre una risposta prima di inviarne una seconda.
Il socket del canale lato Server è di tipo ROUTER:

```
public class zeroMQRouterChannel implements ListenChannel
```

per poter gestire più richieste da più Client deve essere asincrono e deve memorizzare la provenienza dei messaggi: utilizziamo a questo proposito gli ZMsg (vedi ZMQ doc).
Queste classi sono gestite da una classe singleton, *EntitiesManager,* che ha il ruolo di creare e fornire le istanze dei canali e quelle dei Client e del Serversingeton; inoltre questa detiene per ogni Client la lista delle richieste pendenti delle quali non è ancora stata elaborata la risposta fornita dal Server.

Interfaccia *Auth* ed *entities package*

```
public interface Auth {
    //Request
    List<Resource> getListRes();
    Key getAuthentication(String userID,Date deadline,int lvl);
    Token getAuthorization(Key key, String resourceName);
    long checkToken(Token token);
    void clearAuth(int i);
    //Notifications
    void deleteRes(Resource res);
    void addRes(Resource res);
}
```

Quest'ultima espone gli unici metodi invocabili dal lato Client ed implementati dalla classe *ServerSingleton,* il quale adempie, dunque, alle funzioni dell'Autenticatore richiesto:

```
public class ServerSingleton implements Auth {

    private static ServerSingleton instance = null;

    private List<Resource> resourcesList = new ArrayList<~>();
    private List<User> usersList = new ArrayList<~>();
    private List<Token> tokensList = new ArrayList<~>();
```

- *getListRes()* ritorna la lista delle risorse accessibili (*resourceLIst*)
- *getuthentication(),* dato lo userID, una scadenza e livello, torna un oggetto *Key* rappresentante la chiave univoca segreta.

```
public class Key {

    private String keyID;
    private int level;
    private Date deadline;
```

- *getAuthorization(),* data una chiave valida e il nome di una risorsa presente nella lista, ritorna un oggetto di tipo *Token*

```
public class Token {

    private String tokenID;
    private Date deadline;
    private String userID; //owner
    private String resourceID;
```

- *checkToken(),* passato un oggetto *Token* torna un long che rappresenta in millisecondi l'intervallo di tempo rimanente alla scadenza del token (positivo se token non scaduto, viceversa negativo)
- *clearAuth(),* passato l'indice dello oggetto *User* presente nella lista delle autorizzazioni (*userList*), lo rimuove dalla lista.

```
public class User {

    private String userID;
    private Key userKey;
    private int level;
    private List<Token> accessList;
```

- *add/deleteRes()* aggiunge/rimuove l'oggetto *Resource* nella/dalla lista delle risorse (*resourceLIst*)

```
public class Resource {

    private String name;
    private int level;
    private String pathURL;
```

N.B. questi ultimi sono gli unici due metodi invocabili tramite una notifica, la quale non necessita di una risposta e quindi i metodi di un valore di ritorno.
La gestione dei parametri e dei valori di ritorno è affidata alla classe *RequestManager* che analizzeremo in seguito.

Gestione della messaggistica

Sia lato Client - per la creazione di richieste e l'elaborazione di risposte - che lato Server - per l'elaborazione delle richieste e la creazione di risposte - l'applicativo sfrutta la classe singleton *RequestManager,* la quale svolge 3 funzioni fondamentali:
-discrimina l'entità dei messaggi (stringhe json-rpc) in arrivo
-elabora le richieste e le risposte
-crea richieste, risposte ed errori trasformandole in stringhe json-rpc pronte da essere inviate.
Utilizzo della classe lato Client:

```java
public  static class ClientReqThread extends Thread {
    @Override
    public void run() {

        EntitiesManager.getZeroMQReqChannel().connect( ip: "localhost",(short)5555);

        while(true){
            System.out.println("Choose what you want me to do: ");
            System.out.println("- getList");
            System.out.println("- getAuthentication");
            System.out.println("- getAuthorization");
            System.out.println("- checkToken");
            System.out.println("- clearAuth");
            System.out.println("- exit");

            Scanner scanner = new Scanner(System.in);
            String request = scanner.next();
            switch (request){
                case "getList":
                    EntitiesManager.getZeroMQReqChannel().send(RequestManager.getInstance().reqListResource());
                    String response = EntitiesManager.getZeroMQReqChannel().receive();
                    RequestManager.elaborateResponse(response);
                    break;
```

RequestManager viene invocata per tornare la stringa json-rpc rappresentante la richiesta *getList()* da inviare tramite il canale e per elaborare la stringa di risposta ricevuta.
Utilizzo della classe lato Server:

```java
public  static class ServerThread extends Thread {
    @Override
    public void run() {

        EntitiesManager.getZeroMQRouterChannel().bind( ip: "*", (short) 5555);
        while (!Thread.currentThread().isInterrupted()) {
            group_4.Pair<String,String> request = EntitiesManager.getZeroMQRouterChannel().receive();
            EntitiesManager.getZeroMQRouterChannel().reply(request.getKey(),RequestManager.batchDiscriminator(request.getValue()));
        }
    }
}
```

RequestManager viene invocata per discriminare il tipo di messaggio appena ricevuto (*batchDiscriminator()*) e internamente elaborarlo.

RequestManager

Essa quindi è la classe addetta al parsing delle stringhe json-rpc e dunque alla serializzazione/deserializzazione dei messaggi in arrivo ed in uscita: ovvero, infine, è la classe utilizzatrice della Library. `import group_4.Library;`

Come anticipato, la classe possiede tutti i metodi di creazione delle singole richieste esposte nell' interfaccia Auth. Un esempio di metodo appena citato è il seguente:

```java
public String reqAuthentication(String userID, int level, long deadline){
    String method = "getAuthentication";

    Elem elem = new Elem();
    elem.add( key: "userID", new Elem(userID));
    elem.add( key: "lvl", new Elem(level));
    elem.add( key: "deadline", new Elem(deadline));

    Request request = new Request(assignID(), method, elem, JSONRPC_VERSION);
    System.out.println(request.toString());
    EntitiesManager.getInstance().getLogsRequest().add(request);
    return Library.reqToJson(request);
}
```

Simile il comportamento dei metodi analoghi.
N.B. la creazione dei parametri da inserire nel campo params di Request, e dunque in quello della stringa json-rpc, avviene attraverso l'utilizzo del tipo Elem illustrato e spiegato nella documentazione della Library.
N.B. la serializzazione di *request* in stringa json-rpc (*Library.reqToJson*) avviene tramite la libreria ed in particolare tramite gli adapter creati nella Library.
N.B. una volta creato l'oggetto *Request,* esso viene aggiunto alla lista di richieste pendenti, ovvero quelle di cui non è ancora stata ricevuta ed elaborata la risposta.
N.B. il metodo *assignID()* assegna un id incrementale per ogni richiesta:

```java
public class RequestManager {

    private static RequestManager instance = null;
    private int idNum = 0;

    private int assignID (){
        int id = this.idNum;
        this.idNum++;
        return id;
    }
```

Altra funzione principale della classe *RequestManager* è quella, passata una stringa json-rpc, appena ricevuta dal Client/Server, di saper discriminare l'entità del messaggio:

```
public static String rpcDiscriminator(String json){
    if(!Utils.isValidJSON(json)){
        return errorCreator( request: null, ErrorType.PARSE_ERROR);
    }

    Request request = Library.parseRequest(json);
    Response response = Library.parseResponse(json);
    ErrorObj errorObj = Library.parseError(json);
    Notification notification = Library.parseNotification(json);

    if (request != null && response == null && errorObj == null && notification == null){ //isRequest
        return elaborateRequest(json, isNotification: false);
    } else if (request == null && response != null && errorObj == null && notification == null) { //isResponse
        return elaborateResponse(json);
    } else if (request == null && response == null && errorObj != null && notification == null) { //isError
        return elaborateResponse(json);
    } else if (request == null && response == null && errorObj == null && notification != null) { //isNotification
        return elaborateRequest(json, isNotification: true);
    } else {
        return errorCreator( request: null,ErrorType.INVALID_REQUEST);
    }
}
```

La deserializzazione della stringa *json* negli oggetti messaggio passa attraverso i relativi adapter costruiti nella libreria, i quali sono progettati con l'obiettivo di tornare il valore null nel caso in cui la stringa json-rpc passata non rispetti tutti i campi dell'oggetto messaggio da deserializzare: in conclusione l'unico oggetto non null - rappresentante sicuramente l'entità della stringa json-rpc - verrà elaborato.
E' stato aggiunto anche un "discriminator" per i batch, il quale per ogni jsonObject -singolo messaggio del Batch, rappresentato come un jsonArray- chiama *rpcDiscriminator* appena illustrato e costruisce il batch di risposta unendo le singole risposte.

Infine la classe *RequestManager* si occupa dell'elaborazione delle richieste/notifiche e delle risposte chiamando i metodi relativi gestendo i parametri da passare agli stessi e i valori di ritorno. Esempio dell'elaborazione della richiesta del metodo *getAuthorization():*

```
} else if (request.getMethod().equalsIgnoreCase( anotherString: "getAuthorization")) {

    Elem keyId = request.getParams().getElemByKey("keyID");
    Elem resource = request.getParams().getElemByKey("resource");

    if(request.getParams().getSetKey().size() !=2 || keyId==null || resource==null) {
        return errorCreator(request,ErrorType.INVALID_PARAMS );
    }

    if(!EntitiesManager.getInstance().getServer().checkResource(resource.getString())){    // check resource
        return errorServerCreator(request, ErrorTypeAuth.NO_RESOURCES.getMsg(), ErrorTypeAuth.NO_RESOURCES.getCode());
    }

    Key key = null;

    for (int i = 0; i < EntitiesManager.getInstance().getServer().getUsersList().size();i++){
        if(keyId.getString().equalsIgnoreCase(EntitiesManager.getInstance().getServer().getUsersList().get(i).getKey().getKeyID())){
            key = EntitiesManager.getInstance().getServer().getUsersList().get(i).getKey();
            break;
        }
    }

    if (key == null){
        return errorServerCreator(request, ErrorTypeAuth.NO_LOGGED.getMsg(), ErrorTypeAuth.NO_LOGGED.getCode());
    }
```

```
    Token token = EntitiesManager.getInstance().getServer().getAuthorization(key, resource.getString());

    if (token == null){
        return errorServerCreator(request, ErrorTypeAuth.ERROR_LVL.getMsg(), ErrorTypeAuth.ERROR_LVL.getCode());
    }

    Elem tokenIDElem = new Elem(token.getTokenID());
    Elem tokenDeadlineElem = new Elem(token.getDeadline().getTime());
    Elem tokenUserIDElem = new Elem (token.getUserID());
    Elem tokenResourceIDElem = new Elem(token.getResourceID());
    Elem result = new Elem();
    result.add( key: "tokenID", tokenIDElem);
    result.add( key: "deadline", tokenDeadlineElem);
    result.add( key: "userID", tokenUserIDElem);
    result.add( key: "resource", tokenResourceIDElem);

    Response respObj = new Response(request.getId(), result, JSONRPC_VERSION);
    return Library.responseToJson(respObj);
```

Individuato il metodo della richiesta vengono estrapolati i parametri di tipo Elem attraverso la chiave relativa assegnata; dopo gli opportuni controlli di errore, vengono costruiti gli oggetti da passare al metodo (in questo caso l'oggetto di tipo *Key*). I campi del valore di ritorno del metodo vengono "disassemblati" in oggetti di tipo Elem; attribuita una chiave a ciascuno dei campi e composti in un unico oggetto Elem, viene costruito l'oggetto di tipo *Response* e serializzato in stringa json-rpc.
Analizziamo l'elaborazione della risposta alla richiesta con metodo *getAuthorization():*

```
EntitiesManager entitiesManager = EntitiesManager.getInstance();
List<Request> logs = entitiesManager.getLogsRequest();
for (int i = 0; i < logs.size(); i++){
    if (logs.get(i).getId().equals((response.getId()))){
        if (logs.get(i).getMethod().equalsIgnoreCase( anotherString: "getAuthentication")) {
```

```
} else if (logs.get(i).getMethod().equalsIgnoreCase( anotherString: "getAuthorization")){
    //
    Token token = new Token(response.getResult().getElemByKey("tokenID").getString(),
            response.getResult().getElemByKey("userID").getString(),
            response.getResult().getElemByKey("resource").getString(),
            response.getResult().getElemByKey("deadline").getNumber().longValue());

    entitiesManager.getClient().setToken(token);

    /* debug */
    System.out.println("Got Token!");

    entitiesManager.getLogsRequest().remove(logs.get(i));  //delete pending requests
    return null;
```

Innanzitutto ci si accerta che la risposta ricevuta sia una risposta ad una richiesta realmente effettuata dal Client ricevente: ovvero viene cercato l'id della risposta fra quelli delle richieste interne alla lista delle richieste pendenti (*logs*) e successivamente si confronta il metodo. Tramite le chiavi vengono estrapolati gli oggetti di tipo *Elem,* rappresentanti i campi degli oggetti interni al campo *Result* della stringa json-rpc (l'oggetto di tipo *Token* in questo caso)*,* i quali vengono costruiti deserializzandoli. Settati gli opportuni attributi del Client e/o svolte le operazioni che necessitano degli oggetti appena costruiti, si procede all'eliminazione della richiesta -relativa alla risposta appena elaborata- dalla lista delle richieste pendenti.

Gestione degli errori

Oltre alla gestione degli errori di formato json-rpc, eseguita in dalla Library durante la serializzazione/deserializzazione e dai metodi della classe *RequestManager*, il progetto gestisce una serie di errori interni classificati dalla classe enum *ErrorTypeAuth:*

```
public enum ErrorTypeAuth {
    NO_LIST_RES(-32010,"No trace of resources list in memory \n - server"),
    NO_LOGGED(-32020,"KeyID not found. First you need to authenticate"),
    NO_RESOURCES(-32030, "No trace of resource in memory \n - server"),
    ERROR_LVL(-32040, "You're not authorizated"),
    USER_NOT_FOUND(-32060,"No trace of your key in memory"),
    LVL_NOT_VALID(-3270, "Level is not valid"),
    TIME_MACHINE(-32100, "This is not a time machine. You can only choose days after this moment"),
    TOKEN_NOT_FOUND(-32080, "Token not found");
```

Gli errori di formato json-rpc e quelli appena illustrati vengono trasformati in oggetti di tipo ErrorObj rispettivamente dai due metodi *errorCreator() ed errorServerCreator()* interni alla classe *RequestManager.*
N.B. si può costruire un oggetto di tipo ErrorObj sia passando come parametro un oggetto di tipo ErrorType (classe enum interna alla libreria), sia passando codice e messaggio di qualsiasi altra classe enum errore (come ErrorTypeAuth)

```
private static String errorCreator(Request request, ErrorType errorType) {
    if (request == null) {
        ErrorObj error = new ErrorObj( id: "null", errorType, JSONRPC_VERSION);
        return Library.errorToJson(error);
    }
    if (request.getId() != null) {
        ErrorObj error = new ErrorObj(request.getId(), errorType, Constants.JSONRPC_VERSION);
        return Library.errorToJson(error);
    } else {
        System.out.println(" error to who? (id null)");
        return null;
    }
}
```

I messaggi di errori vengo gestiti nel metodo *elaborateResponse()* semplicemente stampando a video il codice di errore e il messaggio.

Guida all'utente

Il progetto sarà provvisto di quattro applicativi Java (.jar) distinti. Oltre al jar della libreria, ci sarà ***authenticator.jar*** che, una volta avviato, risponderà a tutte le richieste ricevute. Una volta avviato il jar dell'autenticatore si dovrà avviare il ***client.jar*** che vi mostrerà una serie di comandi da poter lanciare.

Per un testing più rapido e completo, il quarto jar è chiamato ***test.jar*** e avvia sia l'autenticatore che il client contemporaneamente. Come per il client.jar, vi verrà mostrato a terminale una serie di comandi da poter lanciare, finalizzati al conseguimento delle specifiche fornite dal progetto.

Per l'avvio degli applicativi jar, bisogna, innanzitutto, aver installato Java sulla propria macchina (possibilmente l'ultima versione) e da terminale, entrare nella cartella del file .jar e lanciare il comando

$ java -jar nomefile.jar

N.B. Il client.jar invierà richieste solo in localhost poiché si presume che server e client girino sulla stessa macchina.