UNIVERSITÀ DEGLI STUDI DI MESSINA

PROGETTO DI

PROGRAMMAZIONE 2

SICUREZZA DEI SISTEMI

Vitaliy Lyaskovskiy

472981

INTRODUZIONE ED ANALISI DEGLI STRUMENTI UTILIZZATI

Nella realizzazione di questo progetto ho deciso di utilizzare Java come linguaggio di programmazione, questa scelta mi ha consentito di utilizzare il paradigma della programmazione ad oggetti per realizzare un programma con architettura client-server. Per ottimizzare la gestione delle dipendenze ho utilizzato Maven, questo strumento grazie alla sua capacità di gestire le dipendenze attraverso il file pom.xml, ha semplificato notevolmente il processo di integrazione delle varie librerie. Un aspetto importante del progetto è stata la gestione delle configurazioni esterne per la quale ho preferito l'uso di file XML, questa scelta mi ha permesso di apportare modifiche ai parametri di configurazione senza avere la necessità di modificare il codice. Per l'analisi e il recupero dei dati da questi file XML ho utilizzato l'API DOM di Java. Un altro componente chiave del mio progetto è stato l'impiego di un sistema di logging, l'utilizzo di tale sistema mi ha permesso ti tenere traccia degli errori verificatisi nel server oltre a registrare tentativi di accesso al sistema non autorizzati. La scelta di utilizzare Java mi ha anche permesso e facilitato una gestione granulare degli errori evitando di bloccare l'esecuzione del server al verificarsi di interruzioni dovute alla caduta del client oppure da altri errori relativi alla connessine.

DESCRIZIONE DEL PROGRAMMA

Il progetto che ho sviluppato ha lo scopo di creare un software capace di gestire e archiviare in tempo reale i dati provenienti da un sensore di un misuratore di pressione sanguigna da braccio, questo dispositivo è specificamente progettato per monitorare il battito cardiaco degli utenti. Una componente fondamentale del mio programma è stata assicurare un elevato livello di sicurezza sia nella fase di autenticazione al servizio sia nella conservazione dei dati. Per raggiungere questo obiettivo e proteggere le delicate informazioni degli utenti, ho integrato un efficace sistema di autenticazione a due fattori rafforzando così la sicurezza contro accessi non autorizzati. Per incrementare la robustezza del programma ho dedicato particolare attenzione alla validazione degli input inseriti dagli utenti, questo aspetto si è concentrato principalmente nel verificare che i campi di input non siano vuoti o nulli. Questo tipo di controllo assicura che il software non proceda con dati mancanti o non validi, riducendo il rischio di errori o comportamenti imprevisti durante l'esecuzione. Scendendo nel dettaglio del mio progetto ho implementato un meccanismo di autenticazione che richiede agli utenti di inserire username e password per comunicare con il server, questo processo rappresenta il primo livello di sicurezza del sistema di autenticazione a due fattori. Per la verifica delle credenziali, il server si avvale di un file dedicato alle password dove sono conservate le credenziali degli utenti. Ho scelto di utilizzare un file per le password anziché un database per le credenziali poiché il database in uso nel mio progetto è ottimizzato principalmente per l'archiviazione di record temporali e non per la gestione delle credenziali degli utenti. Nonostante ciò la sicurezza delle informazioni è assicurata grazie all’utilizzo della

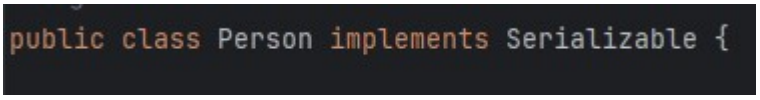
classe BCrypt, questo approccio garantisce che nel file delle password siano memorizzate solo gli hash delle password generati grazie anche all'utilizzo di un "sale" e non le password in chiaro impedendo quindi la lettura diretta delle credenziali. Per il secondo livello di autenticazione ho utilizzato una soluzione basata su One Time Password sincronizzate con il tempo comunemente conosciute come Time Based One Time Password (T-OTP), questo approccio si integra con l'app Google Authenticator installata direttamente sugli smartphone degli utenti. Quando un utente tenta di accedere al sistema, oltre a fornire il proprio username e password, è richiesto di inserire un codice OTP generato dall'app Authenticator. Tale codice, che cambia ogni 30 secondi, è generato in base ad una chiave segreta condivisa tra il server e l'app Authenticator dell'utente. La sincronizzazione basata sul tempo garantisce che ogni codice sia unico. La classe TwoFactorAuthentication del mio progetto gestisce la generazione delle chiavi segrete, la creazione dei codici OTP e la generazione di codici QR che permettono agli utenti di configurare l'app Authenticator semplicemente scansionando il QR code. Per concludere posso affermare che il sistema di autenticazione a due fattori appena descritto combina qualcosa che l'utente conosce (la propria password) con qualcosa che l'utente possiede (il proprio smartphone con l'app Authenticator), fornendo quindi un livello di sicurezza particolarmente elevato che si traduce in un significativamente basso rischio di accessi non autorizzati. L'affidabilità della comunicazione tra client e server è garantita dall'utilizzo del protocollo TCP assicurando che le informazioni inviate e ricevute siano complete e prive di errori. Il server ricopre un ruolo centrale nel programma, è progettato per operare in modalità multithread dove ogni connessione in entrata viene gestita da un thread separato consentendo

così di servire più client in modo efficiente e simultaneo. Per quanto riguarda la memorizzazione dei dati il server si connette a InfluxDB, un database ottimizzato per la gestione di record temporali, questa scelta è stata dettata dalla necessità di avere una soluzione di archiviazione dati che fosse allo stesso tempo veloce, scalabile e adatta a gestire grandi volumi di dati in tempo reale. Infine ho posto grande attenzione alla gestione degli errori all'interno del server, questo include il trattamento di eccezioni come interruzioni di connessione inaspettate, errori nella lettura/scrittura dei dati e problemi di comunicazione con InfluxDB. Il sistema di logging, implementato tramite la classe MyLogger, gioca un ruolo chiave in tale gestione in quanto consente di registrare e analizzare in dettaglio qualsiasi anomalia o malfunzionamento, favorendo così un rapido intervento per la risoluzione dei problemi.

IMPLEMENTAZIONE ED ANALISI DEL CODICE

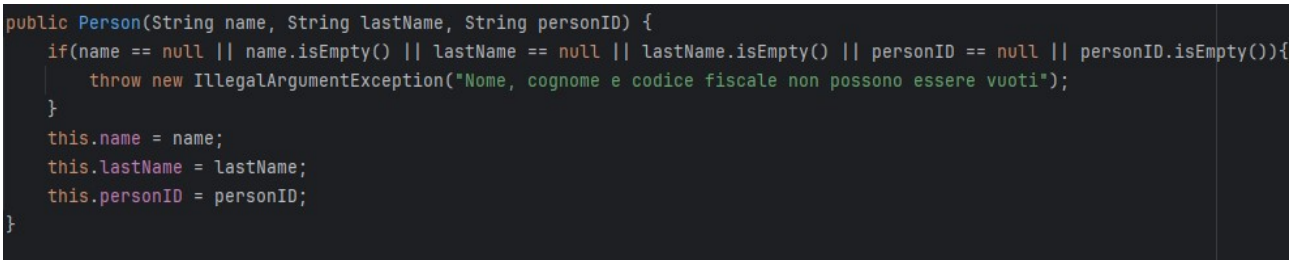
In questa sezione della relazione verranno descritte le componenti principali del codice del mio progetto.

La **classe Person** rappresenta l'identità degli utenti e presenta le seguenti caratteristiche:



```
public class Person implements Serializable {
```

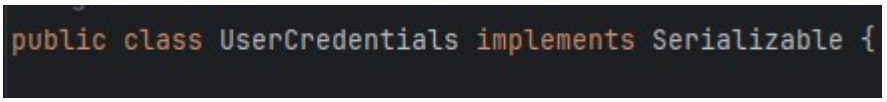
La classe implementa l'interfaccia Serializable, questo consente agli oggetti di tipo Person di essere serializzati e deserializzati.



```
public Person(String name, String lastName, String personID) {
    if(name == null || name.isEmpty() || lastName == null || lastName.isEmpty() || personID == null || personID.isEmpty()){
        throw new IllegalArgumentException("Nome, cognome e codice fiscale non possono essere vuoti");
    }
    this.name = name;
    this.lastName = lastName;
    this.personID = personID;
}
```

All'interno del costruttore e dei metodi setter sono presenti controlli per assicurare che i valori di nome, cognome e codice fiscale siano validi (non nulli e non vuoti).

La classe **UserCredentials** rappresenta l'identità degli utenti e presenta le seguenti caratteristiche:



```
public class UserCredentials implements Serializable {
```

La classe implementa l'interfaccia Serializable, questo consente agli oggetti di tipo UserCredentials di essere serializzati e deserializzati.

```java
public void saveUser(String fileName) {
    try (BufferedWriter writer = new BufferedWriter(new FileWriter(fileName,  append: true))) {
        writer.write( str: username + ":" + BCrypt.hashpw(password, BCrypt.gensalt()));
        writer.newLine();
    } catch (IOException ex) {
        throw new RuntimeException("Errore durante la scrittura delle credenziali", ex);
    }
}
```

Il metodo saveUser ha la funzione di registrare le credenziali degli utenti formate da nome utente (username) seguito da un hash della password e separate da "due punti" (:). L'hash della password è generato utilizzando BCrypt, una funzione di hash che include anche un "sale" per una maggiore sicurezza. In caso di problemi durante la scrittura del file, come un errore di input/output, il metodo solleva un'eccezione segnalando un malfunzionamento nella registrazione delle credenziali.

```java
public static boolean authenticateUser(UserCredentials userToCheck, String fileName) throws IOException {
    try (BufferedReader reader = new BufferedReader(new FileReader(fileName))) {
        String line;
        while ((line = reader.readLine()) != null) {
            String[] parts = line.split( regex: ":");
            if (parts.length == 2) {
                String username = parts[0];
                String hashedPassword = parts[1];

                if(username.equals(userToCheck.getUsername())&&BCrypt.checkpw(userToCheck.getPassword(),hashedPassword)){
                    return true;
                }
            } else {
                throw new IOException("Contenuto del file non valido.");
            }
        }
    } catch (IOException ex) {
        throw new IOException("Non è stato possibile leggere il file delle credenziali", ex);
    }
    return false;
}
```

Il metodo authenticateUser è progettato per verificare se un determinato utente ha diritto di accedere al sistema, quando chiamato legge un file specificato dal nome fileName il quale contiene le credenziali degli utenti registrati. Per ogni riga nel file il metodo divide la stringa in username e password hashata utilizzando il carattere "due punti" (:) come separatore,

successivamente confronta l'username fornito con quelli nel file e, in caso di corrispondenza, verifica se la password fornita corrisponde all'hash salvato tramite la funzione BCrypt.checkpw. In caso di errori nel formato del file o di problemi nel leggerlo il metodo solleva un'eccezione IOException, questa eccezione non viene gestita all'interno del metodo ma viene propagata al chiamante, in questo caso il server, dove può essere registrata e gestita nel sistema di logging. Questo approccio permette una gestione centralizzata degli errori.

La classe **BloodPressureMonitor** è stata realizzata per simulare un misuratore di pressione sanguigna da braccio, implementa caratteristiche già osservate nelle classi precedentemente descritte.

La classe **TwoFactorAuthentication** implementa il secondo fattore di autenticazione:

```
public static String generateToken() {
    SecureRandom random = new SecureRandom();
    byte[] bytes = new byte[20];
    random.nextBytes(bytes);
    Base32 base32 = new Base32();
    return base32.encodeToString(bytes);
}
```

Il metodo generateToken genera un token per l'autenticazione. Utilizza SecureRandom per generare un array di 20 byte casuali e poi lo converte in una stringa usando la codifica Base32.

```
public static String getTOTPCode(String token) {
    Base32 base32 = new Base32();
    byte[] bytes = base32.decode(token);
    String hexKey = Hex.encodeHexString(bytes); //CONVERSIONE IN UNA STRINGA ESADECIMALE
    return TOTP.getOTP(hexKey); //OTP BASATO SULL'ORA CORRENTE E SULLA CHIAVE SEGRETA
}
```

Il metodo getTOTPCode genera un codice OTP (One-Time Password) basato sul tempo. Riceve un token in formato stringa, prima lo decodifica in un array di byte usando Base32, successivamente converte questi byte in una stringa esadecimale la quale verrà poi usata per generare un codice OTP temporaneo tramite la libreria TOTP.

```
public static String getGoogleAuthenticatorBarCode(String token, String account, String issuer) {
    return "otpauth://totp/" //PROTOCOLLO DI AUTENTICAZIONE - TIPO PROTOCOLLO
            + URLEncoder.encode( s: issuer + ":" + account, StandardCharsets.UTF_8).replace( target: "+",  replacement: "%20")
            + "?secret=" + URLEncoder.encode(token, StandardCharsets.UTF_8).replace( target: "+",  replacement: "%20")
            + "&issuer=" + URLEncoder.encode(issuer, StandardCharsets.UTF_8).replace( target: "+",  replacement: "%20");
}
```

Il metodo getGoogleAuthenticatorBarCode crea un URL utile per configurare un account di autenticazione a due fattori nell'app Google Authenticator. Prende come input un token, il nome dell'account e il nome dell'emittente; i valori vengono codificati per essere compatibili con i requisiti del codice QR. L'URL prodotto rispetta il formato del protocollo di Google otpauth://totp/.

```
public static void createQRCode(String barCodeData, String filePath, int height, int width) {
    try {
        BitMatrix matrix = new MultiFormatWriter().encode(barCodeData, BarcodeFormat.QR_CODE, width, height);
        try (FileOutputStream out = new FileOutputStream(filePath)) {
            MatrixToImageWriter.writeToStream(matrix,  format: "png", out);
        } catch (IOException ex) {
            throw new RuntimeException("Errore durante la scrittura del QR code", ex);
        }
    } catch (WriterException ex) {
        throw new RuntimeException("Errore durante la codifica del QR code", ex);
    }
}
```

Il metodo createQRCode genera un QR code. Riceve in input la barCode precedentemente creata, il percorso del file, l'altezza e la larghezza dell'immagine QR. Viene utilizzato MultiFormatWriter per convertire i dati del QR code in un oggetto BitMatrix il quale ne rappresenta la struttura astratta. Infine avviene la scrittura dell'oggetto BitMatrix in formato PNG generando quindi l'immagine del QR code.

La classe **BPMPoint** ha il solo scopo di fornire un metodo statico per la creazione di un punto che successivamente verrà registrato nel database InfluxDB.

```java
public <T> MyLogger(Class<T> tipoClasse) {
    this.logger = Logger.getLogger(tipoClasse.getName());
    // ISTANZA DI LOGGER ASSOCIATA AL NOME DELLA CLASSE PASSATA COME PARAMETRO
}

1 usage
public void setupLogger(String logFileName) {
    try {
        // RIPRISTINO DELLA CONFIGURAZIONE PREDEFINITA
        LogManager.getLogManager().reset();
        logger.setLevel(Level.ALL); // IMPOSTAZIONE DEL LIVELLO DI LOGGING

        FileHandler fileHandler = new FileHandler(logFileName, append: true);
        fileHandler.setFormatter(new SimpleFormatter());

        logger.addHandler(fileHandler); //AGGIUNGE FILEHANDLER AL LOGGER
    } catch (IOException ex) {
        //INTERROMPE L'AVVIO DEL SERVER SE IL LOGGER NON PUO' ESSERE CONFIGURATO
        throw new RuntimeException("Impossibile configurare il logger", ex);
    }
}
```

La classe MyLogger è progettata per gestire la registrazione degli eventi che interessano una classe, nel mio caso il server. Viene inizializzato con il nome che corrisponde al nome della classe, Il metodo setupLogger configura il logger per scrivere i log su un file il cui nome è specificato dal parametro logFileName. Vengono registrati tutti i livelli di log, dal più dettagliato al più generale, ed infine viene utilizzato un FileHandler per scrivere i log con un formato semplice sul file precedentemente specificato. Se si verifica un errore durante la configurazione del logger, ad esempio un problema nell'accesso al file di log, il metodo solleva un'eccezione

interrompendo l'esecuzione del codice, questo per evitare l'avvio del server senza un adeguato sistema di logging.

La classe **MyClient** rappresenta il client in una comunicazione con architettura client-server:

```
private static void loadConfigurations() {
    try (FileInputStream file = new FileInputStream( name: "configClient.xml")){

        // PARSING DEL DOCUMENTO XML
        DocumentBuilderFactory dbFactory = DocumentBuilderFactory.newInstance();
        DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
        Document doc = dBuilder.parse(file);

        //ESTRAZIONE DEI DATI DAL DOCUMENTO XML
        doc.getDocumentElement().normalize();
        Node serverNode = doc.getElementsByTagName("client").item( index: 0);
        if (serverNode.getNodeType() == Node.ELEMENT_NODE) {
            Element serverElement = (Element) serverNode;
            serverName = serverElement.getElementsByTagName("serverName").item( index: 0).getTextContent();
            serverPort = Integer.parseInt(serverElement.getElementsByTagName("serverPort").item( index: 0).getTextContent());
        } else {
            throw new IllegalStateException("Il nodo 'client' nel file XML non è un elemento");
        }
    } catch (Exception ex) {
        throw new RuntimeException("Errore durante il caricamento del file di configurazione del Client", ex);
    }
}
```

Il programma inizia caricando la configurazione dei dati da un file esterno in formato XML, viene utilizzato DOM come parser per estrarre i dati necessari.

```
try (
        Socket socket = new Socket(serverName, serverPort);
        ObjectOutputStream out = new ObjectOutputStream(socket.getOutputStream());
        ObjectInputStream in = new ObjectInputStream(socket.getInputStream())
) {
```

Successivamente stabilisce una connessione TCP con il server utilizzando blocchi try-with-resources. Dopo aver stabilito la connessione procede con una fase di autenticazione a due fattori: richiede prima le classiche credenziali e solo in caso di verifica positiva procede con la verifica del codice OTP. Una volta autenticato viene simulato l'invio di dati al server utilizzando le classi precedentemente descritte.

La classe **MyServer** rappresenta il server in una comunicazione con architettura client-server:

```
loadConfigurations();

logger = new MyLogger(MyServer.class);
logger.setupLogger(logFileName);

try (ServerSocket serverSocket = new ServerSocket(port)) {

    logger.getLogger().info( msg: "Server in ascolto sulla porta: " + port);

    System.out.println("Server in ascolto sulla porta: " + port); // DEBUG

    while (true) {
        Socket clientSocket = serverSocket.accept();
        System.out.println("Connessione stabilita con il client: " + clientSocket.getInetAddress().getHostAddress()); //DEBUG
        logger.getLogger().info( msg: "Connessione stabilita con il client: " + clientSocket.getInetAddress().getHostAddress());
        new ClientHandler(clientSocket).start();
    }
} catch (IOException ex) {
    logger.getLogger().log(Level.SEVERE,  msg: "Errore nella creazione della socket del server", ex);
    throw new RuntimeException("Errore nella creazione della socket nel server", ex);
}
```

Inizialmente il server carica le configurazioni utilizzando il metodo loadConfigurations(), successivamente configura un logger per la registrazione degli eventi su un file di log. Una volta configurato, il server avvia un ServerSocket sulla porta specificata e attraverso un ciclo infinito, resta in attesa di connessioni dai client. Quando un client si connette il server ne registra l'indirizzo ip, successivamente delega la gestione di questa connessione a un nuovo ClientHandler seguendo l'approccio del multithreading, infatti ogni ClientHandler potrà occuparsi di un singolo client permettendo quindi al server di gestire più connessioni contemporaneamente. Se si verifica un errore nell'apertura della socket, il server registra un messaggio di errore grave (severe) nel logger e solleva una RuntimeException indicando un problema critico nell'avvio del server.

```
try (InfluxDBClient influxDBClient = InfluxDBClientFactory.create(influxDB_URL, influxDB_Token.toCharArray(), organization, bucket)) {

    WriteApiBlocking writeApiBlocking = influxDBClient.getWriteApiBlocking();

    Object receivedObject;
```

In questa parte del codice il server stabilisce una connessione con il database InfluxDB utilizzando i parametri precedentemente caricati dal file di configurazione.

```
while (true) {
    try {
        receivedObject = in.readObject();

        if (receivedObject instanceof BloodPressureMonitor) {
            BloodPressureMonitor bpMonitor = (BloodPressureMonitor) receivedObject;
            writeApiBlocking.writePoint(BPMPoint.createBPMPoint(bpMonitor));
            System.out.println("Punto scritto nel database"); //DEBUG
        }

        if (receivedObject instanceof String && receivedObject.equals("END")) { //CONTROLLO DEL SEGNALE DI FINE TRASMISSIONE
            out.writeObject("Ricezione dei dati conclusa - Connessione Terminata");
            break;
        }

    } catch (EOFException ex) {
        logger.getLogger().log(Level.SEVERE,  msg: "Connessione interrotta prematuramente dal client", ex);
        out.writeObject("Ricezione dei dati interrotta - Connessione Terminata");
        break; // INTERROMPE IL CICLO PER EOF
    } catch (SocketException ex) {
        logger.getLogger().log(Level.SEVERE,  msg: "Comunicazione interrotta improvvisamente dal client", ex); //Crash del client
        break;
    }
}
```

Nella sua parte centrale il server utilizza un ciclo while per continuare a ricevere i dati dal client e salvarli nel database, questa fase si ripete fino a quando verrà inviato un messaggio “END” da parte del client che indica di aver terminato l’invio dei dati. Il server inoltre gestisce e registra nel log anche una possibile caduta del client oppure una chiusura anticipata della connessione. Per concludere il logger viene utilizzato, oltre che per registrare malfunzionamenti, anche per tenere traccia di tentativi falliti di connessine dovuti all’inserimento di credenziali oppure codici OTP errati.