

# LABORATORIO INTERDISCIPLINARE B

Corso di Laurea in Informatica – Progetto Java

Anno Accademico 2021 – 2022



## Manuale Tecnico

Vellone Alex 741527 Varese

Macaj Manuel 741854 Varese

Said Ibrahim Mahdi 741512 Varese

Pazienza Silvio 741486 Varese



# Sommario

- Contesto .....3**
- Struttura:.....3**
- Scelte algoritmiche:.....4**
- Diagrammi:.....5**
- Sicurezza:.....9**
- Scelte architetturali.....10**
  - Compilazione..... 10
  - Database ..... 10
- Strutture dati .....11**
- Pattern .....12**

## Contesto:

L'obiettivo del progetto Centri Vaccinali è quello di realizzare un sistema di gestione per centri vaccinali basato su sistemi di registrazione di centri vaccinali e utenti che possono associarsi a qualsiasi di questi. In seguito alla registrazione e alla vaccinazione ogni cittadino può segnalare una reazione avversa al vaccino utilizzando una scala di valutazione da 1 a 5 ed un commento supplementare di massimo 256 caratteri.

Il progetto Centri Vaccinali è composto da due applicazioni che lavorano sullo stesso database ma con funzionalità diverse. Queste sono accessibili solo da due tipologie di utenti: operatori o cittadini.

L'applicazione **Cittadini** consente di:

- Visualizzare l'elenco dei centri vaccinali
- Filtrare l'elenco dei centri vaccinali in base a: nome centro vaccinale, comune, tipologia
- Visualizzare il dettaglio completo di un centro vaccinale
- Finalizzare il processo di registrazione presso un centro vaccinale di propria scelta
- Visualizzare i commenti e l'indice di gravità sulle reazioni avverse dagli utenti registrati
- Scrivere i dettagli per un eventuale evento avverso per un centro vaccinale, solo se si è loggati presso di esso.

L'applicazione **CentriVaccinali**, accessibile solo agli operatori, consente di:

- Creare e registrare un nuovo centro vaccinale da aggiungere all'elenco;
- Inizializzare il processo di registrazione di un cittadino presso un centro vaccinale di sua scelta.

**Entrambe** le applicazioni necessitano di una connessione con l'applicazione server per funzionare.

## Struttura:

Il progetto è stato sviluppato in un gruppo di 4 persone con Java 11 utilizzando la libreria grafica Swing. Per gestire il nostro progetto abbiamo deciso di creare un repository privato su GitHub dove poter condividere, in modo rapido e sicuro, il codice e avere sempre a disposizione la cronologia del nostro sviluppo.

Sempre su GitHub abbiamo creato tutti i task necessari per completare il progetto, in modo da poterci dividere il lavoro in modo equo.

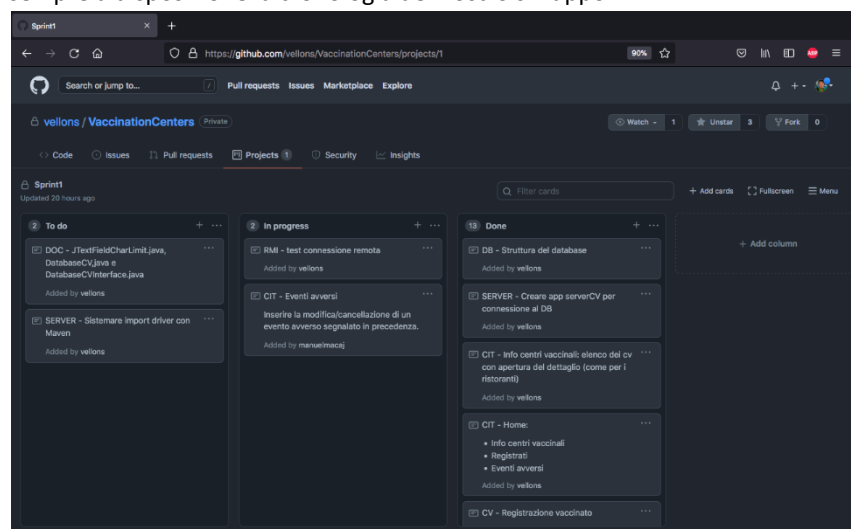
Così facendo ognuno di noi sapeva sempre lo stato di avanzamento di tutti i task.

Tutti i dati delle applicazioni sono salvati su un database remoto.

Le informazioni rimangono quindi esterne al computer nel quale viene eseguita l'applicazione.

Le classi *Vaccinato*, *CentroVaccinale* e *EventoAvverso* implementavano la classe Java *Serializable*. La quale offre un meccanismo che permette di rappresentare i dati in una sequenza di byte contenente tutte le sue informazioni più importanti, come le variabili ed il loro tipo.

Tutte le variabili delle classi che abbiamo creato sono definite *private* in modo da garantire l'integrità dei dati e per proteggerli da manipolazioni accidentali.



## Scelte algoritmiche:

Per poter gestire concorrentemente molteplici istanze dei programmi “Cittadini” e “CentriVaccinali” è stata predisposta un’applicazione “Server” per gestire la comunicazione con il database.

Il metodo utilizza la Java RMI (Remote Method Invocation), una tecnologia caratteristica del linguaggio Java che consente ai processi di invocare metodi su classi remote disponibili su un server.

Esempio (Connessione con il Server):

```
try {
    // Recupero dell'IP del registry server se nelle variabili d'ambiente
    if (System.getenv("CV_SERVER_HOST") != null) {
        REGISTRY_SERVER_IP = System.getenv("CV_SERVER_HOST");
    }

    // Richiesta dell'indirizzo ip o hostname del server all'utente
    while (REGISTRY_SERVER_IP == null) {
        String ip = JOptionPane.showInputDialog(null, "Inserisci l'IP del
server:", "localhost");
        if (ip != null && !ip.isEmpty()) {
            if (isValidIP(ip) || ip.equals("localhost")) {
                REGISTRY_SERVER_IP = ip; // Se l'indirizzo è valido posso
proseguire
            } else {
                System.err.println("IP " + ip + " non riconosciuto. Inserisci un
IPv4 valido");
            }
        }
    }

    // Creazione del registry RMI
    Registry reg = LocateRegistry.getRegistry(REGISTRY_SERVER_IP,
REGISTRY_SERVER_PORT);
    database = (DatabaseCVInterface) reg.lookup("CVDatabaseServer");
    database.logMessage("Nuovo client connesso");
}
```

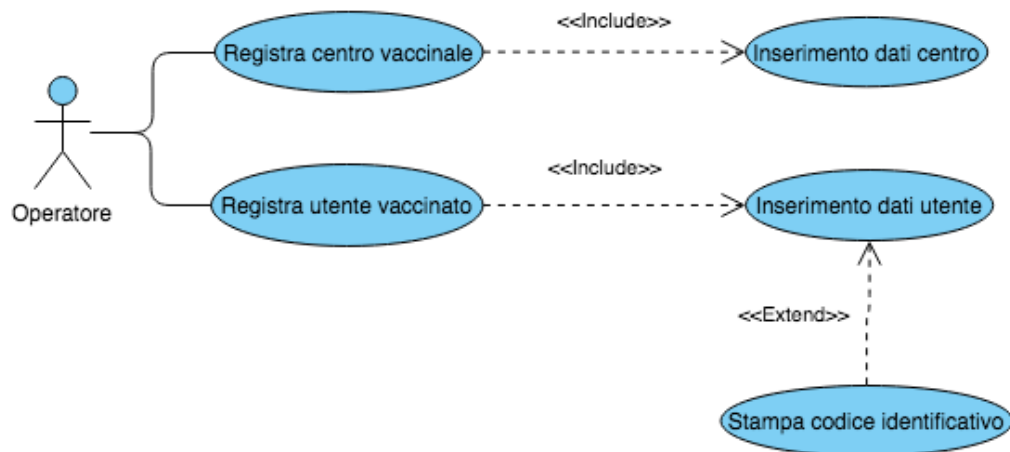
- Se tra le variabili d’ambiente viene trovato l’indirizzo IP del server, questo viene automaticamente impostato, in caso contrario spetterà all’utente inserirlo manualmente.
- È anche possibile connettersi localmente tramite l’indirizzo impostato di default qualora non vi fossero variabili d’ambiente registrate.
- In ogni caso, se l’indirizzo inserito è corretto, questo viene salvato.
- Successivamente viene creato un’ interfaccia remota Registry con i parametri di indirizzo IP e numero di porta (quest’ultimo col valore predefinito 1099).
- La funzione di lookup restituisce il riferimento remoto al registro che sarà usato dal database.

## Diagrammi:

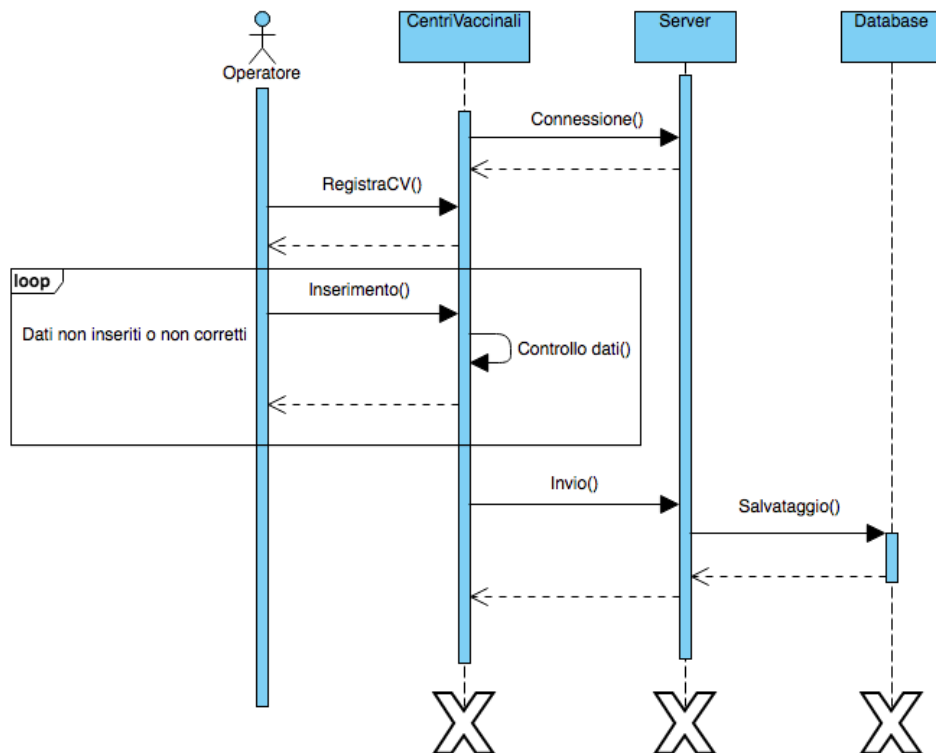
Quelli mostrati di seguito sono i diagrammi UML ed il diagramma Entità-Relazione che definiscono l'applicazione.

All'interno di questi diagrammi si possono vedere come sono utilizzati i vari package, le classi che sono state implementate e tutte le variabili che vengono usate.

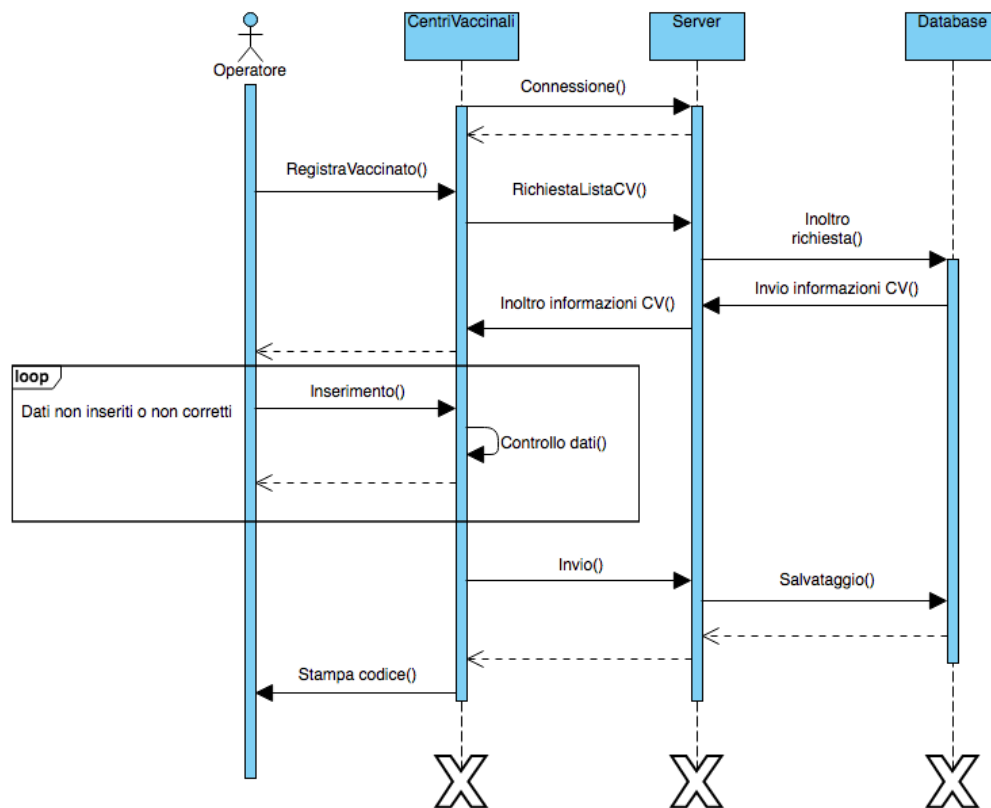
**Diagramma dei casi d'uso (lato operatori):**



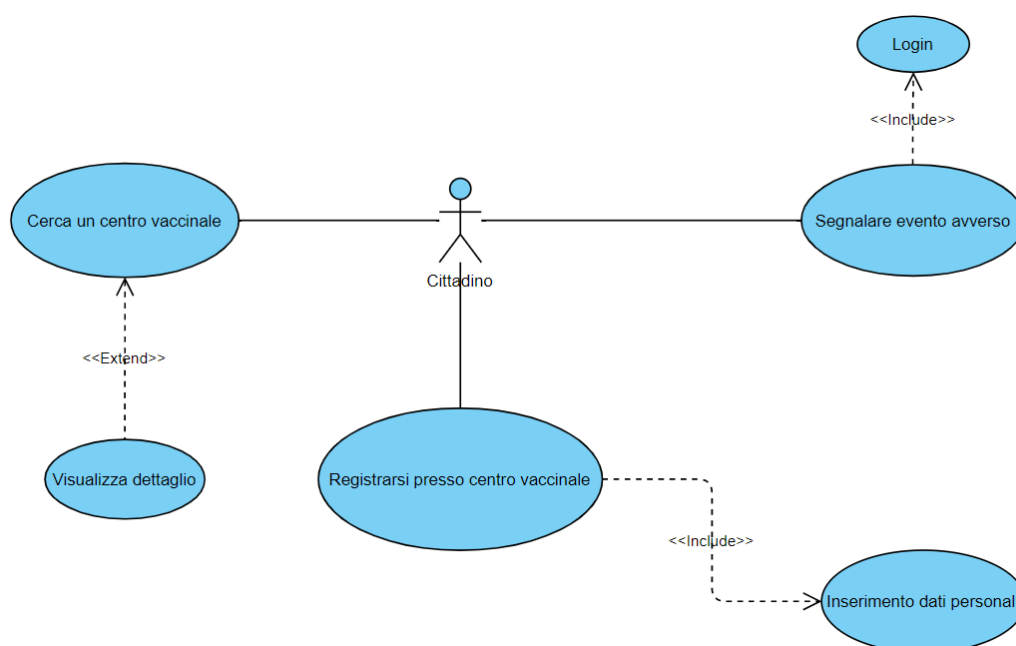
**Diagramma di sequenza - Registrazione centro vaccinale:**



### Diagramma di sequenza - Registrazione utente vaccinato:

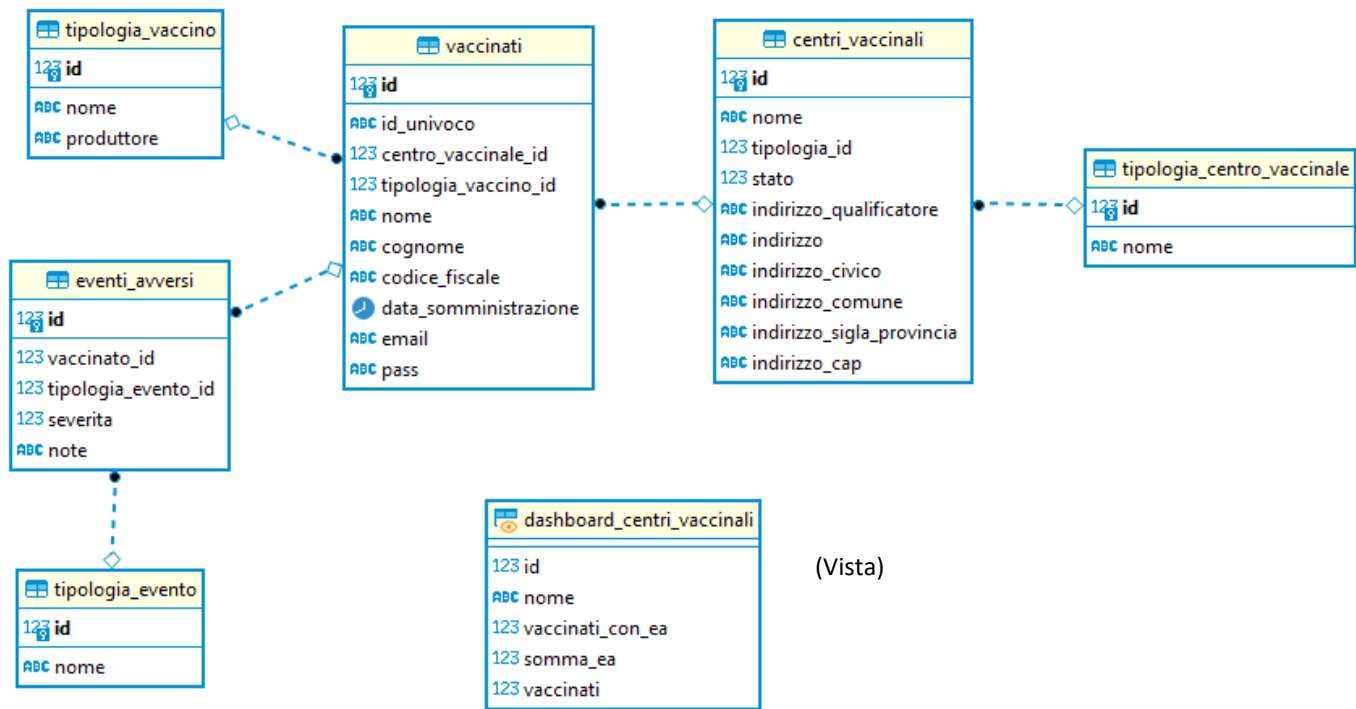


### Diagramma dei casi d'uso (lato cittadini):



[illegible]

Diagramma Entità-Relazione:





## Sicurezza:

L'applicazione non salva mai la password degli utenti in chiaro. Appena riceve il testo della password si preoccupa di calcolare la firma della stringa utilizzando l'algoritmo **Sha1**. La firma della password verrà poi salvata sul database.

Quando l'utente si vorrà loggare all'interno dell'interfaccia Swing verrà istanziato un oggetto DatabaseCV che cercherà nel database l'email e una firma uguale alla firma della password che ha inserito l'utente. Se dopo che è stato applicato il filtro è presente un'utenza, le credenziali saranno confermate e l'utente potrà procedere con le operazioni.

Esempio (login):

```
public void actionPerformed(ActionEvent e) {

    lblErrors.setText("");
    utenteLoggato = null;
    if (tfEmail.getText() == null || tfEmail.getText().length() < 3 ||
    tfPassword.getText() == null || tfPassword.getText().length() < 3) {
        lblErrors.setFont(new Font("Default", Font.BOLD, 14));
        lblErrors.setText("Completare i campi correttamente");
        lblErrors.setForeground(Color.RED);
        return;
    }

    try {
        DatabaseCVInterface db =
        ServerConnectionSingleton.getDatabaseInstance(); // Singleton class con il
        server

        Vaccinato v =
        db.getVaccinatoByEmailAndPasswordSha(tfEmail.getText(), tfPassword.getText());
        if (v == null) {
            lblErrors.setFont(new Font("Default", Font.BOLD, 14));
            lblErrors.setText("Email o password errati");
            lblErrors.setForeground(Color.RED);
        } else {
            utenteLoggato = v;
            tfEmail.setText("");
            tfPassword.setText("");
            openDashElencoEventiAvversi();
        }
    } catch (Exception ex) {
        lblErrors.setText("Ci sono stati problemi con il login. Prova a
        riavviare l'app");
        ex.printStackTrace();
        ServerConnectionSingleton.resetConnection();
    }
}

});
```

- Inizialmente si verifica che tutti i dati siano conformi a quanto accettato dal database, in caso affermativo viene creato un oggetto DatabaseCV usando una connessione Singleton (più dettagli in seguito)
- Viene creato un nuovo oggetto Vaccinato, dopodiché vengono prelevati dal database i dati dell'utente

# Scelte architetturali

## Compilazione

Per lo sviluppo di Centri Vaccinali è stato utilizzato IntelliJ IDEA come IDE. Questa applicazione è stata sviluppata con Java 11 e per essere eseguita è necessario che sia installata una versione di Java 11 o superiore.

Per poter compilare l'applicazione è necessario avere installato sul proprio computer la JDK di Java e **Maven**. Maven è uno strumento di gestione per i progetti software, che ci permette di compilare il progetto e quindi creare degli eseguibili .jar da poter condividere con gli utenti e operatori sanitari utilizzatore dell'applicazione.

Utilizzando IntelliJ IDEA come IDE per la programmazione verrà installato automaticamente Maven. Per compilare il progetto ed ottenere i file eseguibili .jar bisognerà utilizzare un terminale ed eseguire il comando: *mvn clean compile package install*. I 3 file eseguibili .jar (server, centri vaccinali e cittadini) si troveranno nella cartella target a fine compilazione.

Quando si utilizza il terminale di IntelliJ IDEA bisognerà premere CTRL+INVIO per eseguire il comando di compilazione Maven.

Maven viene utilizzato anche per scaricare le dipendenze del progetto:

- Postgresql: libreria che si occupa della gestione del database (JDBC)
- Jcommon e jfreechart: libreria utilizzata per creare dei grafici

## Database

Per la realizzazione di questo progetto si è utilizzato un database Postgresql. Per garantire a questa applicazione un'elevata affidabilità e bassi tempi di attesa, è stato deciso di ospitare il database utilizzando un servizio **database-as-a-service**. Il database di questa applicazione viene fornito da [elephantsql](#).

Con elephantsql possiamo scegliere tra diversi piani a pagamento che variano per: dimensione del database, connessione contemporanee e numero di repliche dei dati. Nel nostro caso abbiamo utilizzato l'unico piano gratuito disponibile. Nel momento in cui l'applicazione andrà in produzione si potrà iniziare ad utilizzare un piano a pagamento e aumentare la dimensione del database.

Elephantsql permette inoltre di scegliere dove ospitare il database tra diverse regioni di: Google Cloud, Amazon AWS e Microsoft Azure.

Nel nostro caso il database è ospitato da **Amazon Web Services** nella regione **eu-west-1**.

Elephantsql fornisce dominio, username e password per la connessione al database.

L'accesso al database avviene utilizzando il JDBC (Java Database Connectivity) di Postgresql.

Abbiamo deciso di ospitare il nostro database su elephantsql perché:

- Il database è ospitato in un datacenter a bassissima latenza
- È scalabile: se le informazioni aumentano si può cambiare il piano e richiedere più spazio
- È stato possibile scegliere dove salvare i dati. Nel nostro caso i dati rimangono salvati in Europa. Questo facilita il trattamento dei dati personali degli utenti (GDPR)


Durante dei test preliminari abbiamo constatato che il tempo di esecuzione di una query di lettura sul database è di circa 55mS. L'applicazione server mostra, per ogni query eseguita, il tempo necessario per ottenere la risposta del database.

TINY TURTLE

Shared high performance server

20 MB data

5 concurrent connections



**FREE**

GET STARTED

Try now for FREE

## Strutture dati

L'applicazione server si occupa di mantenere la connessione con il database e permette, ai client connessi, di eseguire operazioni concorrenti di lettura e scrittura sul database.

L'applicazione server, una volta avviata, presenta tre input text e due bottoni. I tre input text servono per inserire server, username e password del database, mentre i due pulsanti permettono di connettersi o disconnettersi dal database.

*Metodo per la connessione al database (file Server.java):*

```
private void connectToDB() {
    try {
        Class.forName("org.postgresql.Driver");
    } catch (java.lang.ClassNotFoundException e) {
        logMessage("ERROR: Driver mancante: " + e.getMessage());
    }

    String url = "jdbc:postgresql://" + host + "/" + DB_NAME;

    try {
        conn = DriverManager.getConnection(url, username, password);
        logMessage("Connessione con il database stabilita");
        btnAccedi.setEnabled(false);
        btnDisconnetti.setEnabled(true);
        tfHostDB.setEditable(false);
        tfUsernameDB.setEditable(false);
        tfPasswordDB.setEditable(false);
        dbCV.setConnection(conn);
    } catch (SQLException e) {
        logMessage("ERROR: " + e.getMessage());
    }
}
```

Se la connessione con il database avviene con successo il pulsante per accedere al database verrà disabilitato.

Da quel momento il server potrà eseguire tutte le query richieste dai client in maniera concorrente.

I client si connettono al database sfruttando Java **RMI** (Remote Method Invocation). All'avvio del server parte anche un registry RMI con il nome "CVDatabaseServer", sulla porta 1099. I client che si vorranno connettere dovranno conoscere l'indirizzo IP del server, il nome del registry e la porta RMI per la connessione.

*Server.java: creazione del registry RMI:*

```
// Bind del registry per la connessione con RMI
try {
    dbCV = new DatabaseCV(textAreaServerStatus);
    Registry reg = LocateRegistry.createRegistry(REGISTRY_PORT);
    reg.rebind(REGISTRY_NAME, dbCV);
    logMessage("Registry RMI attivo. Porta=" + REGISTRY_PORT + ". Nome=" +
REGISTRY_NAME);
} catch (Exception e) {
    logMessage("ERROR registry: " + e.getMessage());
    e.printStackTrace();
}
```

I client connessi potranno utilizzare tutti i metodi definiti nell'interfaccia "**DatabaseCVInterface**". Ogni metodo di questa interfaccia esegue un query sul database e restituisce il risultato al client. "DatabaseCVInterface" è implementato nella classe "**DatabaseCV**" che estende la classe "**UnicastRemoteObject**".

*Per esempio, metodo per recuperare l'elenco delle persone vaccinate:*

```
public List<Vaccinato> getVaccinati() throws RemoteException {
    List<Vaccinato> returnList = new ArrayList<>();
    try {
        long startTime = System.nanoTime();
        Statement stmt = conn.createStatement();
        String query = "SELECT * FROM vaccinati;";
        ResultSet rs = stmt.executeQuery(query);
        while (rs.next()) {
            Vaccinato obj = new Vaccinato(rs.getInt("id"));
            obj.setId_univoco(rs.getString("id_univoco"));
            obj.setCentro_vaccinale_id(rs.getInt("centro_vaccinale_id"));
            obj.setTipologia_vaccino_id(rs.getInt("tipologia_vaccino_id"));
            obj.setNome(rs.getString("nome"));
            obj.setCognome(rs.getString("cognome"));
            obj.setCodice_fiscale(rs.getString("codice_fiscale"));
            obj.setEmail(rs.getString("email"));
            obj.setPass(rs.getString("pass"));
            returnList.add(obj);
        }
        rs.close();
        stmt.close();
        long duration = (System.nanoTime() - startTime) / 1000000;
        logMessage(query + " in: " + duration + "mS");
    } catch (Exception e) {
        logMessage("ERROR: getVaccinati()");
        e.printStackTrace();
    }
    return returnList;
}
```

## Pattern

I client (applicazione centri vaccinali e applicazione cittadini) hanno bisogno di recuperare molte informazioni da mostrare agli utenti. Per recuperare queste informazioni del database viene utilizzata una connessione con il server.

La connessione con il server viene gestita attraverso la classe “**ServerConnectionSingleton**” nel package global. Questa classe implementa il design pattern **Singleton** che permette di creare una sola istanza di una classe utilizzabile in punti diversi dell'applicazione.

Nel nostro caso il design pattern Singleton, ci permette di utilizzare la classe per connettersi con il database in più punti nel codice, utilizzando una sola connessione con il server.

*Esempio di utilizzo della classe singleton per recuperare l'elenco dei vaccinati:*

```
// db è il Singleton utilizzato per la connessione con il server
DatabaseCVInterface db = ServerConnectionSingleton.getDatabaseInstance();

List<Vaccinato> vaccinati = db.getVaccinati();
for (Vaccinato v : vaccinati) {
    System.out.println(v.toString());
}
```

Porzione di codice della classe **ServerConnectionSingleton** per la creazione del Singleton e la richiesta dell'indirizzo IP del registry server:

```
public class ServerConnectionSingleton {

    private static ServerConnectionSingleton singleton_instance = null;
    private static DatabaseCVInterface database;
    private static String REGISTRY_SERVER_IP = null;
    private final static int REGISTRY_SERVER_PORT = 1099;

    private ServerConnectionSingleton() {
        try {
            // Recupero dell'IP del registry se nelle variabili d'ambiente
            if (System.getenv("CV_SERVER_HOST") != null) {
                REGISTRY_SERVER_IP = System.getenv("CV_SERVER_HOST");
            }

            // Richiesta dell'indirizzo ip o hostname del server all'utente
            while (REGISTRY_SERVER_IP == null) {
                String ip = JOptionPane.showInputDialog(null, "Inserisci l'IP
del server:", "localhost");
                if (ip != null && !ip.isEmpty()) {
                    if (isValidIP(ip) || ip.equals("localhost")) {
                        REGISTRY_SERVER_IP = ip;
                    } else {
                        System.err.println("IP " + ip + " non riconosciuto.
Inserisci un IPv4 valido");
                    }
                }
            }

            // Creazione del registry RMI
            Registry reg = LocateRegistry.getRegistry(REGISTRY_SERVER_IP,
REGISTRY_SERVER_PORT);
            database = (DatabaseCVInterface) reg.lookup("CVDatabaseServer");
            database.logMessage("Nuovo client connesso");
        } catch (RemoteException | NotBoundException e) {
            System.err.println("ERROR: creazione singleton con il server");
            database = null;
            e.printStackTrace();
        }
    }

    public static DatabaseCVInterface getDatabaseInstance() {
        if (singleton_instance == null || database == null) {
            singleton_instance = new ServerConnectionSingleton();
        }
        return database;
    }
}
```

Per maggiori informazioni sulle classi e sui metodi implementati in questa applicazione è stata creata la JavaDoc che puoi aprire da un qualsiasi browser. Il file per accedervi è disponibile in questa cartella: [javadoc.html](#)

In caso di difficoltà di avvio dell'applicazione “Cittadini” o “CentriVaccinali” ti consigliamo di leggere il manuale utente. Al suo interno è presente una sezione FAQ dove troverai una guida per risolvere gli errori più comuni.