ELABORATO PER LA PROVA D'ESAME

INDIRIZZO: ITIA - INFORMATICA E TELECOMUNICAZIONI
ARTICOLAZIONE "INFORMATICA"
ANNO SCOLASTICO 2019-2020
Discipline: INFORMATICA, SISTEMI E RETI

Sommario

Traccia:

Progettare un sistema che, tramite geolocalizzazione indoor, supporti gli utenti in ambito domestico o all'interno di ambienti pubblici.

In particolare, fatte le opportune ipotesi, si chiede di descrivere:

- il progetto dell'infrastruttura tecnologica ed informatica necessaria a gestire il sistema, dettagliando:
  - l'architettura di rete e le caratteristiche dei sistemi server;
  - le modalità di comunicazione;
  - gli elementi per garantire la sicurezza del sistema;
- l'architettura dei dati e dei software che si intende implementare, dettagliando il modello dei dati e le modalità di interazione;
- una parte significativa dell'applicazione.

Soluzione:

ANALISI

In questo periodo di pandemia è molto importante mantenere il distanziamento fisico. Questo può essere difficile, soprattutto se ci si trova in un ambiente pubblico molto affollato come un supermercato.

Il sistema che ho progettato ha lo scopo di evitare la creazione di assembramenti, permettendo ai gestori di uno spazio pubblico di monitorare in tempo reale le posizioni delle persone e i percorsi effettuati (previo consenso degli utenti). Il sistema in particolare è pensato per essere utilizzato in **negozi e supermercati**, dove i clienti si devono muovere all'interno dei locali e c'è un alto rischio di affollamenti. Il sistema supporta anche catene di supermercati, con diverse sedi, consentendo la gestione delle informazioni in modo centralizzato.

È necessario quindi un sistema che permetta la **geolocalizzazione indoor** degli utenti. Per fare questo escludo l'uso del GPS, in quanto, in ambienti chiusi, non ha la precisione richiesta. Utilizzo invece un sistema di localizzazione basato sulle antenne **Wi-Fi** presenti all'interno della struttura.

Ogni cliente avrà una **scheda intelligente** basata su un microcontrollore e dotata di antenna Wi-Fi, di uno scanner in grado di leggere i codici QR e di uno schermo sul quale vengono visualizzate le informazioni dell'utente che la sta utilizzando.

Al fine di localizzare le schede, devono essere presenti almeno **tre access-point Wi-Fi**. Ogni scheda misura costantemente la potenza del segnale proveniente dalle tre antenne Wi-Fi e la invia a un server. Il server poi si occupa di calcolare l'effettiva distanza da ogni access-point e applica un algoritmo di **trilaterazione** per trovare la posizione dell'utente.

Si crea inoltre un **sito web** utilizzabile sia dai clienti sia dai gestori dell'ente.

Il personale, dopo aver effettuato il login, può gestire le schede e visualizzare in tempo reale gli spostamenti delle persone su una mappa.

Gli utenti invece, tramite il sito web, hanno la possibilità registrarsi al sistema, inserendo le loro informazioni personali. Alla fine della registrazione, viene generato un codice QR contenente un identificativo univoco. Al momento dell'accesso al luogo pubblico, l'utente ritira una scheda e scannerizza il suo codice, in modo da **associarla** al suo profilo. Da questo momento la scheda è attiva e la sua posizione viene costantemente salvata. I gestori dell'ente hanno inoltre la possibilità di associare manualmente dal sito web una scheda a un

determinato cliente, in caso di problemi con il codice QR. Dopo aver effettuato il login al sito, gli utenti possono visualizzare lo storico dei loro utilizzi delle schede e i percorsi da loro effettuati.

È possibile inoltre realizzare un'applicazione per smartphone che consenta una miglior fruizione del servizio da dispositivi mobili. In questo progetto verrà implementata solo la versione web in quanto, tramite dei framework come Bootstrap, si possono realizzare siti web mobile responsive, utilizzabili anche da dispositivi mobili senza nessun problema e senza dover obbligatoriamente installare un'applicazione nativa.

È necessario quindi un server che si occupi dello scambio d'informazioni tra le schede intelligenti e il sito web e del calcolo della posizione.

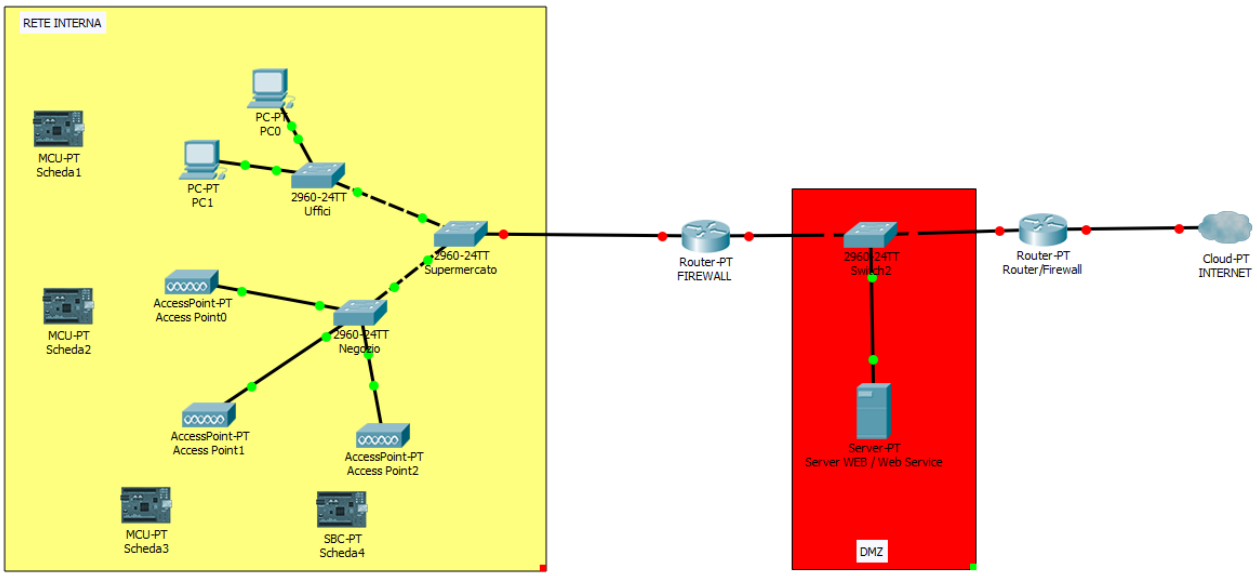
ARCHITETTURA DI RETE

Il sistema ha un'architettura "**three-tier**": viene utilizzato un DBMS a cui solo il server ha accesso, mentre i client (le schede e il sito web) accedono al database passando per l'interfaccia esposta dal server.

Per il server abbiamo due possibilità:

- acquistare un server fisico
- utilizzare un servizio **cloud**

Nel **primo** caso l'architettura del sistema sarebbe la seguente:



È prevista la presenza di un server su cui risiedono il sito web e il database. Il server ha un sistema operativo basato su Linux. Si utilizza MySQL per la gestione del database e Apache Tomcat per il web service e il sito web.

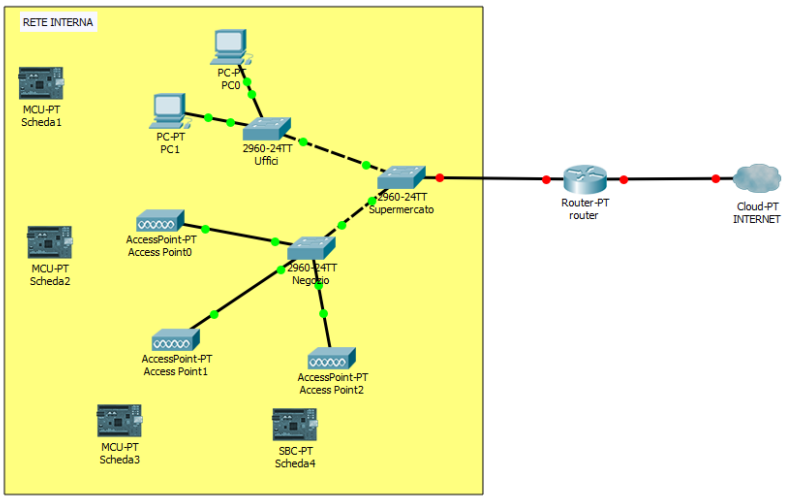
L'introduzione di un server locale, però, porterebbe a dover gestire tutte le problematiche relative alla sicurezza e alla possibile perdita dei dati.

Il server infatti dovrebbe essere accessibile dall'esterno della rete. Si crea quindi una **DMZ** (demilitarized zone, zona demilitarizzata), il cui scopo è quello di contenere i server che devono essere accessibili dall'esterno, in modo da non consentire l'accesso diretto alla rete interna.

La **seconda** possibilità invece è quella che si sta diffondendo sempre di più negli ultimi anni e presenta diversi vantaggi. Primo fra tutti è la **scalabilità**, che può essere verticale o orizzontale. Nel primo caso viene aumentata la capacità di una singola macchina, mentre nel secondo vengono aggiunte nuove macchine su cui distribuire il carico di lavoro, tramite sistemi di **load balancing**.

Scelgo quindi di utilizzare quest'ultima soluzione. In questo modo è possibile incrementare o ridurre le risorse disponibili in base alle necessità del cliente, senza dover spegnere e riaccendere la macchina. È possibile inoltre, utilizzando dei sistemi di **autoscaling**, aumentare o diminuire automaticamente il numero di istanze dei server, in base all'uso corrente delle risorse, in modo da mantenerne le prestazioni stabili in qualsiasi condizione di utilizzo. Inoltre, non abbiamo più il problema di dover effettuare il backup dei dati, in quanto viene gestito dal servizio che acquistiamo.

L'architettura scelta è quindi la seguente:



In questo modo si consente anche l'uso del sistema a quegli enti che non hanno la possibilità di acquistare un server fisico, portando grandi vantaggi in termini di risparmio ed efficienza.

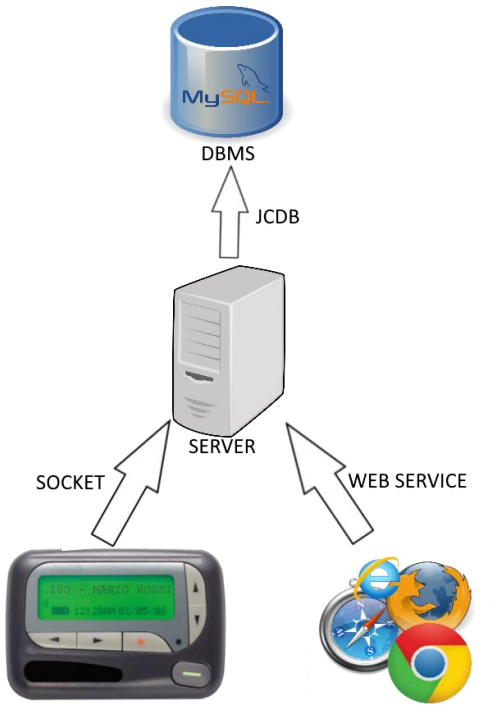
La rete interna è suddivisa in due **sottoreti**, utilizzando le **VLAN**. Una sottorete è una suddivisione creata usando una parte del campo host dell'indirizzo IP. Si utilizza poi la subnet mask per identificare la parte di indirizzo corrispondente alla rete e quella riferita all'host. Per fare questo, i bit della subnet mask corrispondenti alla parte di rete sono messi a 1, mentre quelli corrispondenti alla parte host sono a 0. La tecnologia VLAN consente inoltre di creare reti locali virtuali, non comunicanti tra loro, ma che sfruttano la stessa infrastruttura fisica. In questo modo sarà il router (livello 3) a indirizzare i pacchetti tra una VLAN e l'altra.

Alla prima VLAN sono connessi gli access point, mentre alla seconda i computer degli uffici, utilizzati anche per la gestione delle schede.

Le schede sono connesse alla rete tramite **Wi-Fi**. Dato che la rete Wi-Fi viene utilizzata anche per calcolare la posizione dell'utente, è necessario che la perdita di potenza sia minima. In particolare, bisogna ridurre al minimo lo **slow-fading**, la perdita di potenza dovuta all'assorbimento del segnale da parte degli ostacoli. Questo verrebbe interpretato come una distanza maggiore dall'access point, causando errori nella localizzazione. È indispensabile quindi prestare molta attenzione al posizionamento delle antenne, studiando prima l'ambiente (**adaptive modulation**).

MODALITÀ DI COMUNICAZIONE

Considerando la possibilità dello sviluppo in futuro di applicazioni native per smartphone, scelgo di utilizzare un web service con un **API Rest** per la comunicazione con il sito. In questo modo garantisco l'interoperabilità tra diverse applicazioni e piattaforme. Inoltre, usando un web service basato sul protocollo HTTP, non è necessario effettuare modifiche ai firewall, in quanto è solitamente già consentito. Il sito web in questo modo utilizza i metodi esposti dal web service per l'interazione con il database.



La comunicazione tra il server e le schede avviene invece tramite **socket**, in quanto è necessaria una comunicazione bidirezionale. Il server deve essere infatti in grado di inviare messaggi alle schede, per esempio quando vengono associate a un utente da remoto. Utilizzo il protocollo di livello trasporto **TCP**, che mi garantisce una connessione affidabile. Questo significa che non devo preoccuparmi dell'ordine di arrivo dei pacchetti o di eventuali perdite. La socket è composta da due parti:

- l'indirizzo IP, un indirizzo di 32 bit assegnato al dispositivo interessato;
- la porta, un numero di 16 bit che identifica il processo interessato.

Il database è inoltre configurato per accettare solo connessioni provenienti da "localhost", in quanto è utilizzata un'architettura three-tier.

Per motivi di performance, scelgo di utilizzare il formato **JSON** per lo scambio dei dati tra il server e il sito web. A differenza di XML infatti, è molto più leggero e viene processato molto più velocemente, grazie alla sua struttura.

Per la comunicazione tra il server e le schede utilizzo invece un formato **CSV**. Questo perché è necessario ridurre al minimo la dimensione dei pacchetti, per evitare di sovraccaricare la rete wireless. I formati XML e JSON introdurrebbero infatti un overhead dovuto alla struttura dei dati.

SICUREZZA DEL SISTEMA

Come anticipato, la rete interna è divisa in sottoreti. Per garantire la sicurezza, la sottorete che contiene le schede può comunicare solo con il server. Questo obiettivo può essere raggiunto tramite le **ACL**, liste ordinate di regole che filtrano il traffico. Un esempio di ACL è il seguente:

```
access-list 100 permit tcp <rete schede> <wildcard> host <ip server> eq 5000
access-list 100 permit tcp host <ip server> <rete schede> <wildcard> established
access-list 100 deny ip <rete schede> <wildcard> any
access-list 100 deny ip any <rete schede> <wildcard>
```

Le prime due righe consentono la comunicazione TCP tra la sottorete contenente le schede e il server sulla porta 5000. Le ultime due righe vietano qualsiasi altro tipo di comunicazione alla sottorete.

Per garantire la **sicurezza** del web service si utilizza inoltre un **token JWT**. Questo token viene generato dal server al momento del login e inserito nei cookies e nell'authorization header. I client poi dovranno includerlo in ogni successiva richiesta.

L'API Rest infatti, essendo stateless, non memorizza informazioni sullo stato dei client. All'interno del token sono quindi codificate tutte le informazioni necessarie per identificare l'utente che l'ha generato: l'id, lo username e una variabile booleana che identifica i gestori.

Il token generato viene firmato digitalmente con una **chiave privata**, conosciuta solo dal server, per evitare che un malintenzionato possa modificarlo e fingersi un'altra persona. In questo modo vengono garantite sia **l'autenticità** del server, sia **l'integrità** delle informazioni contenute nel token.

Anche utilizzando un token per l'autenticazione, tuttavia, c'è ancora un problema di sicurezza. I pacchetti infatti, viaggiando in chiaro attraverso la rete internet, potrebbero essere facilmente intercettati e manomessi.

Per ottenere una maggior sicurezza, il server utilizza quindi una connessione **HTTPS** (HTTP over TLS). Grazie a questo protocollo, sono garantite:

- l'**autenticità** del server. Il server infatti ha un **certificato** digitale firmato da un'autorità conosciuta, che ne attesta la vera identità;
- la **riservatezza** delle informazioni e **l'integrità** dei dati. Grazie al protocollo TLS (successore del SSL), il payload del pacchetto HTTP viene infatti criptato prima di essere inviato. In questo modo l'URL, i parametri, gli headers e i cookies vengono cifrati, mentre l'indirizzo IP e la porta viaggiano in chiaro (in quanto sono rispettivamente a livello 3 e 4).

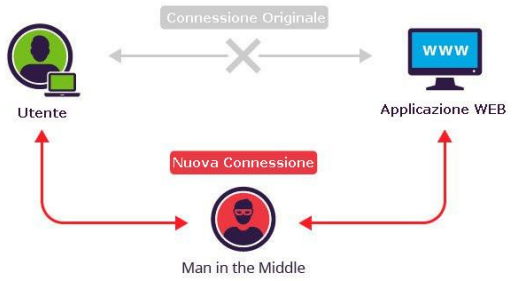
La crittografia usata dal protocollo **TLS** utilizza chiavi simmetriche. Lo scambio delle chiavi avviene tramite un processo di **handshake** durante il quale:

- il server invia al client il suo certificato;
- il client verifica il certificato ed estrae la chiave pubblica del server;
- il client genera una chiave casuale di sessione e la cifra con la chiave pubblica del server;
- il server, con la sua chiave privata, decifra il messaggio del client contenente la chiave da usare per la comunicazione.

Da questo momento in poi la comunicazione sarà cifrata con una chiave simmetrica di sessione.

Il protocollo HTTPS crea quindi un canale di comunicazione sicuro attraverso una rete non sicura. Ciò garantisce una **protezione accettabile**, anche se non totale, da alcune tipologie di attacchi:

- MITM (man in the middle), un utente malintenzionato intercetta e ritrasmette la comunicazione tra due parti, che credono di comunicare direttamente tra di loro;
- eavesdropper (intercettazione), la comunicazione fra due o più interlocutori viene ascoltata da terzi;
- session hijacking (dirottamento di sessione), i cookies usati per autenticare l'utente vengono rubati e utilizzati per effettuare un accesso non autorizzato.



Per mitigare gli effetti di un eventuale data breach, inoltre, le password non vengono salvate in chiaro nel database, ma si utilizza una funzione di **hash** one way (irreversibile) per cifrarle. Così facendo, anche se venissero rubate, sarebbero solo delle stringhe di caratteri incomprensibili. Risalire alla password originale sarebbe possibile solo nel caso di una stringa nota, altrimenti sarebbe necessario un attacco di tipo brute force, che richiederebbe molto tempo e risorse.

ARCHITETTURA DEI DATI

Per la gestione delle informazioni utilizzo un **DBMS**, un sistema software che gestisce dei dati strutturati. Un DBMS quindi deve:

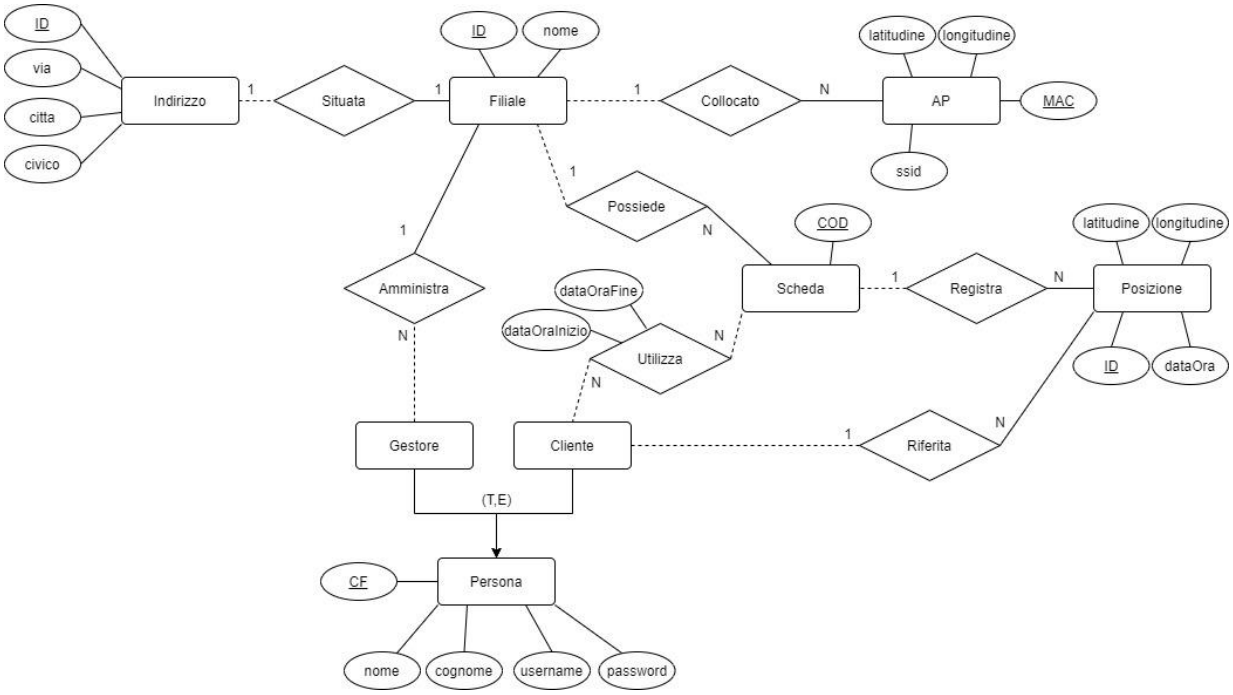
- garantire la **persistenza** dei dati, conservando intatto il contenuto della base di dati anche attraverso un sistema di backup e ripristino;
- garantire la **consistenza** dei dati, evitando stati di inconsistenza dovuti ad accessi concorrenti;
- supportare le **transazioni**, sequenze di operazioni che possono concludersi con il successo di tutte o con insuccesso. In quest'ultimo caso non deve rimanere traccia delle operazioni effettuate;
- garantire **l'integrità** dei dati, applicando i vincoli imposti ai tipi di dati;
- garantire la **sicurezza** dei dati tramite un sistema di autenticazione e autorizzazione;
- gestire i metadati, le informazioni che descrivono la base di dati.

Il DBMS inoltre deve essere efficace ed efficiente.

È necessario quindi progettare l'architettura della base di dati. Le **entità** che emergono sono le seguenti:

- "*Scheda*", l'entità centrale che rappresenta una scheda intelligente;
- "*Posizione*", rappresenta una posizione registrata da una scheda;
- "*Filiale*", rappresenta una sede, nel caso per esempio di catene di supermercati;
- "*Indirizzo*", rappresenta l'indirizzo di una filiale. Non utilizzo un attributo composto nell'entità "*Filiale*" in quanto il database non sarebbe in prima forma normale;
- "*Persona*", comprende i gestori e i clienti. La generalizzazione (ISA) è totale ed esclusiva, in quanto si suppone che, nel caso un gestore sia anche un cliente, abbia due account diversi.

Lo schema ER, che rappresenta il **modello concettuale**, è il seguente:



Le **associazioni** più rilevanti sono:

- "*Utilizza*", quando un cliente ritira una scheda, questa viene associata a lui e vengono registrate ora d'inizio e ora di fine.
- "*Riferita*", indica l'utente a cui è riferita una posizione registrata. Si potrebbe anche utilizzare l'orario per ricavare quest'informazione, ma con un'associazione è presente un legame "forte".

Prima di derivare lo schema logico, effettuo alcune ristrutturazioni:

- la generalizzazione (**ISA**) viene risolta eliminando l'entità "*Persona*" e mantenendo le due entità figlie, in quanto contengono diverse associazioni. Gli attributi dell'entità padre vengono di conseguenza copiati in entrambe le entità figlie;
- l'associazione N-N "*Utilizza*", diventerà anch'essa una relazione;
- le chiavi primarie CF e MAC, vengono sostituite con chiavi artificiali ad autoincremento, per motivi di prestazioni. Le chiavi sostituite e "*username*" sono chiavi candidate e avranno il vincolo di **unique**.

Il **modello logico**, dopo la ristrutturazione, è quindi il seguente:

```
Filiale (ID, nome, idIndirizzo*)
Indirizzo (ID, via, citta, civico)
AP (ID, mac, ssid, latitudine, longitudine, idFiliale*)
Gestore (ID, cf, nome, cognome, username, password, idFiliale*)
Cliente (ID, cf, nome, cognome, username, password)
Utilizza (IDCLIENTE*, CODSCHEDA*, DATAORAINIZIO, dataOraFine)
Scheda (COD, idFiliale*)
Posizione (ID, dataOra, latitudine, longitudine, codScheda*, idCliente*)
```

La chiave primaria di "*Utilizza*" è composta, oltre che dalle due chiavi esterne, dall'ora di inizio, per garantire l'univocità dei record.

La definizione delle relazioni in linguaggio SQL (**DDL**) è la seguente:

```
CREATE TABLE Indirizzo(
    ID INT PRIMARY KEY AUTO_INCREMENT,
    via VARCHAR(32) NOT NULL,
    civico VARCHAR(6) NOT NULL, --può
contenere lettere
    citta VARCHAR(32) NOT NULL
);

CREATE TABLE Filiale(
    ID INT PRIMARY KEY AUTO_INCREMENT,
    nome VARCHAR(32) NOT NULL,
    idIndirizzo INT NOT NULL
        REFERENCES Indirizzo(ID)
            ON UPDATE CASCADE
            ON DELETE CASCADE --Se elimino la
filiale, elimino anche il suo indirizzo
);

CREATE TABLE Cliente(
    ID INT PRIMARY KEY AUTO_INCREMENT,
    cf CHAR(16) UNIQUE NOT NULL,
    nome VARCHAR(32) NOT NULL,
    cognome VARCHAR(32) NOT NULL,
    username VARCHAR(32) UNIQUE NOT NULL,
    password CHAR(32) NOT NULL --hash MD5
);

CREATE TABLE Gestore(
    ID INT PRIMARY KEY AUTO_INCREMENT,
    cf CHAR(16) UNIQUE NOT NULL,
    nome VARCHAR(32) NOT NULL,
    cognome VARCHAR(32) NOT NULL,
    username VARCHAR(32) UNIQUE NOT NULL,
    password CHAR(32) NOT NULL, --hash MD5
    idFiliale INT REFERENCES Filiale(ID)
        ON UPDATE CASCADE
        ON DELETE SET NULL
);
```

```
CREATE TABLE AP(
    ID INT PRIMARY KEY AUTO_INCREMENT,
    mac CHAR(12) UNIQUE NOT NULL,
    ssid VARCHAR(32) NOT NULL,
    latitudine DECIMAL(7,5) NOT NULL
CHECK(latitudine BETWEEN -90 AND 90),
    longitudine DECIMAL(7,5) NOT NULL
CHECK(longitudine BETWEEN -180 AND 180),
    idFiliale INT NOT NULL
        REFERENCES Filiale(ID)
            ON UPDATE CASCADE
            ON DELETE CASCADE
);

CREATE TABLE Scheda(
    COD INT PRIMARY KEY AUTO_INCREMENT,
    idFiliale INT NOT NULL
        REFERENCES Filiale(ID)
            ON UPDATE CASCADE
            ON DELETE SET NULL
);

CREATE TABLE Utilizza(
    idCliente INT NOT NULL
        REFERENCES Cliente(ID)
            ON UPDATE CASCADE
            ON DELETE SET NULL,
    codScheda INT NOT NULL
        REFERENCES Scheda(COD)
            ON UPDATE CASCADE
            ON DELETE SET NULL,
    dataOraInizio DATETIME NOT NULL,
    dataOraFine DATETIME, --può essere nullo
(se è ancora in uso)

    PRIMARY KEY (idCliente, codScheda,
dataOraInizio)
);
```

```
CREATE TABLE Posizione(
    ID INT PRIMARY KEY AUTO_INCREMENT,
    dataOra DATETIME NOT NULL,
                                --imposto dei vincoli per latitudine e longitudine
    latitudine DECIMAL(7,5) NOT NULL CHECK(latitudine BETWEEN -90 AND 90),
    longitudine DECIMAL(7,5) NOT NULL CHECK(longitudine BETWEEN -180 AND 180),
    codScheda INT NOT NULL
        REFERENCES Scheda(COD)
            ON UPDATE CASCADE
            ON DELETE SET NULL,
    idCliente INT NOT NULL
        REFERENCES Cliente(ID)
            ON UPDATE CASCADE
            ON DELETE SET NULL
);
```

Presento ora alcune delle **query** principali per il corretto funzionamento del sistema.

1. Registrare una nuova posizione:

```
INSERT INTO Posizione(latitudine,longitudine,dataOra,idScheda)
   VALUES ([latitudine], [longitudine], [dataOra], [idScheda]);
```

2. Visualizzare l'elenco delle schede usate da un cliente

```
SELECT codScheda, dataOraInizio, dataOraFine FROM Cliente
   INNER JOIN Utilizza ON (Cliente.id = Utilizza.idCliente)
WHERE Cliente.id = [idCliente];
```

3. Visualizzare il percorso effettuato (elenco di coordinate) dalla scheda in un determinato giorno (indipendentemente dal cliente)

```
SELECT latitudine, longitudine, dataOra FROM Posizione
    INNER JOIN Scheda ON (Scheda.COD = Posizione.codScheda)
WHERE date(dataOra) = [dataScelta];
```

4. Visualizzare il percorso effettuato (elenco di coordinate) da un determinato cliente

```
SELECT latitudine, longitudine, dataOra FROM Posizione
    INNER JOIN Scheda ON (Scheda.COD = Posizione.codScheda)
    INNER JOIN Utilizza ON (Scheda.COD = Utilizza.codScheda)
WHERE Utilizza.idCliente = [idCliente];
```

5. Visualizzare il cliente che ha usufruito più volte del servizio

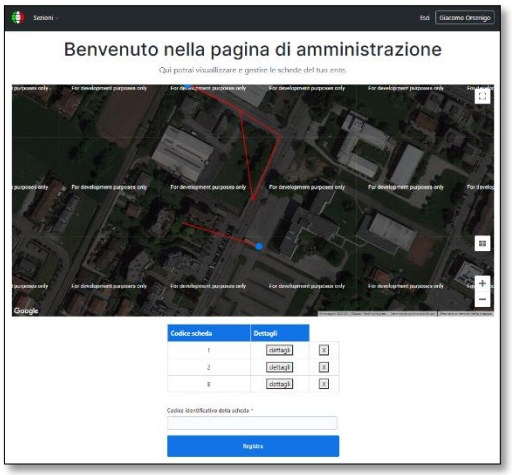
```
SELECT Cliente.* FROM Cliente
    INNER JOIN Utilizza ON (Cliente.id = Utilizza.idCliente)
GROUP BY idCliente
HAVING count(*) >= ALL (SELECT count(*) FROM Utilizza
                        GROUP BY idCliente);
```

6. Visualizzare i clienti che non hanno mai usufruito del servizio

```
SELECT * FROM Cliente
WHERE id NOT IN (SELECT DISTINCT idCliente
                 FROM Utilizza);
```

PARTE SIGNIFICATIVA DELL'APPLICAZIONE

Il sito web è realizzato con **HTML e JavaScript**. Per i clienti sono disponibili le pagine di registrazione, di login e una homepage in cui si può visualizzare l'elenco delle schede utilizzate con i relativi percorsi da loro effettuati. I gestori invece, come si può vedere nell'immagine a lato, hanno la possibilità di visualizzare le informazioni di tutti i clienti, compresa una mappa con i percorsi effettuati, e di gestire le schede disponibili.



Per effettuare le richieste al web service si utilizza **AJAX**, che permette di fare chiamate remote asincrone, caricando dinamicamente la pagina.

Di seguito è mostrato un estratto della pagina di login. Questa pagina effettua una richiesta di tipo GET e, in caso di successo, reindirizza l'utente alla homepage.

```
<html>
   <head>
      <meta name="viewport" content="width=device-width, initial-scale=1">
      <link rel="stylesheet" type="text/css" href="stileRegistrazione.css">
      <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.4.1/jquery.min.js"></script>
      <script>
        function login() {
            $.ajax({
                url: "/api/users?username=" + $("#username").val() + "&password=" +
$("#password").val(),
                type: 'GET',
                xhrFields: {
                    withCredentials: true
                },
            }).done((data) => {  //l'autenticazione è riuscita e il token è salvato nei cookies
                location.href = "home.html";
            }).fail((jqXHR, textStatus, errorThrown) => {
                console.error(errorThrown);
                if (jqXHR.status == 401) //credenziali errate
                    $("#error").html("Credenziali errate");
                else
                    $("#error").html("Errore durante l'autenticazione");
            });
        }
      </script>
  </head>
  <body>
     <form method="GET" onsubmit="return login()">
         <font color="red" id="error"></font>
         <label for="username">Username</label>
         <input type="text" placeholder="username" name="username" id="username" required/>
         <label for="password">Password</label>
         <input type="password" placeholder="Password" name="password" id="password" required/>
         <input type="submit" value="Accedi"/>
     </form>
  </body>
</html>
```

Il server, dopo aver effettuato la richiesta AJAX, risponderà inserendo il token in un cookie. Nel caso si volessero estrarre le informazioni dal token, per esempio per mostrare lo username dell'utente attualmente autenticato o per controllare il livello di autorizzazione, si può utilizzare la libreria jwt-decode. In questo modo è possibile solamente estrarre le informazioni contenute nel token; sarà poi compito del server, al momento dell'invio di una richiesta, controllarne la validità.

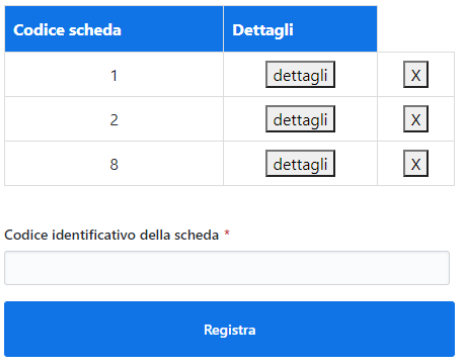
Presento ora il codice della tabella di gestione delle schede, tramite la quale il personale addetto può visualizzare le schede disponibili, eliminarle o visualizzarne una pagina di dettagli. La tabella è caricata dinamicamente quando viene ottenuta la risposta dal web service.

```
<script>
  function confermaEliminazione(cod) {
    if (confirm("Confermi l'eliminazione della scheda?")) {
      $.ajax({
        url: "http://localhost:8080/ProggyWebServices/api/devices/" + cod,  //URL dell'API
        type: 'DELETE', //metodo HTTP
        xhrFields: {
          withCredentials: true  //invio anche il token contenuto nei cookies
        },
      }).done((data, textStatus, jqXHR) => {
        $("#tabella > thead").empty();  //svuoto la tabella e la ricarico
        loadSchede();
      }).fail((jqXHR, textStatus, errorThrown) => {
        $("#errors").html("Dissociazione non riuscita.<br>" + errorThrown + "<br>Assicurati che il Web Service sia attivo e
funzionante");
      });
    }
  }

  function loadSchede() {
    $.ajax({
      url: "http://localhost:8080/ProggyWebServices/api/devices", //URL dell'API
      type: 'GET', //metodo HTTP
      contentType: "application/json", //formato dei dati
      xhrFields: {
        withCredentials: true  //invio anche il token contenuto nei cookies
      },
    }).done((data, textStatus, jqXHR) => {
      var json = data;    //l'oggetto in JSON viene automaticamente riconosciuto da JavaScript
      json.devices.forEach(device => {     //Aggiungo ogni riga alla tabella
        var cod = device.cod;
        var row =  $("#tabella > tbody")[0].insertRow(-1);
        row.insertCell(0).innerHTML = cod;
        row.insertCell(1).innerHTML = "<a href=\"dettaglioScheda.html?idScheda=" + cod +
          "\"><button>dettagli</button></a>";
        row.insertCell(2).innerHTML = "<button onclick=\"confermaEliminazione(" + cod + ")\">X</button>";
      })
    }).fail((jqXHR, textStatus, errorThrown) => {
      $("#errors").html("Caricamento non riuscito.<br>" + errorThrown + "<br>Assicurati che il Web Service sia attivo e
funzionante");
    });
  }

  $(document).ready(function () {
    loadSchede();
  });
</script>

<table id="tabella">
  <thead>
    <tr>
      <th>Codice scheda</th>
      <th>Dettagli</th>
    </tr>
  </thead>
  <tbody><!-- Qui si troveranno i record--></tbody>
</table>
```



La tabella viene quindi compilata con le informazioni ricevute dal web service in formato JSON. L'eliminazione di una scheda è effettuata tramite una richiesta di tipo DELETE. È presente inoltre un piccolo form con il quale è possibile aggiungere una nuova scheda, tramite una richiesta di tipo POST.

Il server invece è realizzato in linguaggio Java. Il server gestisce la comunicazione con il database. A questo scopo si utilizza il connettore **JDBC**. Di seguito riporto la classe che, attraverso il pattern singleton, gestisce la connessione al database. Per la connessione si utilizza un utente apposito, che deve necessariamente avere i permessi di lettura e modifica e una password complessa.

```
public class DbManager {

   private static DbManager INSTANCE;

   private static final String driver = "com.mysql.cj.jdbc.Driver";
   private static final String dbms_url = "jdbc:mysql://localhost/";
   private static final String database = "id12710500_personalsafety_new";
   private static final String user = "server_negozio";
   private static final String password = "N9j1$m6YW%Wz*J2o";
   private Connection connection;
   private boolean connected;

   private DbManager() {
      connected = connect();
   }

   private boolean connect() {
      final String url = dbms_url + database + "?serverTimezone=Europe/Rome";
      try {
         Class.forName(driver);
         connection = DriverManager.getConnection(url, user, password);
         return true;
      } catch (SQLException | ClassNotFoundException e) {
         e.printStackTrace();
         return false;
      }
   }

   public static synchronized DbManager getInstance() {
      if (INSTANCE == null) {
         INSTANCE = new DbManager();
      } else if (!INSTANCE.connected) {
         INSTANCE.connected = INSTANCE.connect();
      }
      return INSTANCE;
   }

   public Connection getConnection() {
      return connection;
   }

   public boolean isConnected() {              //controlla se la connessione è ancora attiva

      try {
         connected = connection.isValid(0);
      } catch (SQLException throwables) {
         throwables.printStackTrace();
      }
      return connected;
   }

}
```

Per lo sviluppo del web service si utilizza **Jersey**, un framework open source che implementa JAX-RS.

Il web service ha quindi un'architettura Restful, ciò significa principalmente due cose:

- è focalizzato sul concetto di risorsa, identificabile attraverso un URI e condivisa attraverso un'interfaccia comune tra client e server;
- è stateless ovvero non mantiene informazioni sullo stato dei client. È quindi necessario implementare un sistema di sessioni, ad esempio attraverso un token JWT.

Presento ora il metodo che gestisce una richiesta di login, è di tipo GET e ha due parametri nella query: username e password.

```
/**
 * Effettua il login con username e password
 * Richiama {@link #authenticate(String, String)}
 *
 * @return token se l'autenticazione è riuscita, {@link Response.Status#UNAUTHORIZED} in caso contrario
 */
@GET
@Produces(MediaType.APPLICATION_XML)
public Response login(@QueryParam("username") String username,
                      @QueryParam("password") String password) {
   try {
      // Authenticate the user using the credentials provided
      // Issue a token for the user
      String token = authenticate(username, password);
      // Return the token on the response
      return Response
            .ok()
            .header(AUTHORIZATION, "Bearer " + token)
            .cookie(new NewCookie("Token", token))
            .build();
   } catch (SecurityException e) {
      e.printStackTrace();
      return Response.status(UNAUTHORIZED).build();
   }
}
```

Il metodo authenticate, dopo aver effettuato le query per verificare la correttezza delle informazioni, restituisce un token generato con il metodo seguente:

```
/**
 * Genera un token con i parametri specificati
 * @param id    id dell'utente
 * @param user  username dell'utente
 * @param admin booleano che indica se l'utente è un ammistratore
 * @return token generato
 * @see Jwts#builder()
 */
public String issueToken(int id, String user, boolean admin, UriInfo context) {
   return Jwts.builder()
         .setSubject(user)              //inserisco le informazioni sull'utente
         .claim("id", id)
         .claim("admin", admin)
         .setIssuer(context.getAbsolutePath().toString())
         .setIssuedAt(new Date())
         .setExpiration(toDate(LocalDateTime.now().plusMinutes(15L)))       //imposto la scadenza
         .signWith(key, SignatureAlgorithm.HS512)       //key è la chiave con cui viene firmato il token
         .compact();
}
```

Nelle successive richieste, il server verifica che sia presente un token e che sia valido. In caso affermativo, la richiesta viene eseguita, in caso negativo viene inviata una risposta con il codice di errore HTTP 401 – non autorizzato. Di seguito il codice che verifica il token contenuto nei cookies:

```
final Cookie cookie = requestContext.getCookies().get(AUTHENTICATION_COOKIE_NAME);
if (cookie != null){
   try {
      //Verifico la validdità del token
      final Claims claims = JwtAuthenticator.getInstance().authenticate(token).getBody();
      System.out.println("#### valid token : " + token);

      final SecurityContext currentSecurityContext = requestContext.getSecurityContext();
      final MyPrincipal utente = new MyPrincipal(claims.getSubject() ,  //estraggo dal token i dati  dell'utente
            claims.get("id", Integer.class),
            claims.get("admin", Boolean.class));
      requestContext.setSecurityContext(new SecurityContext() { //inserisco l'utente nel context della richiesta
         @Override                                              // in modo da poter poi risalire a chi l'ha fatta
         public Principal getUserPrincipal() {
            return utente;
         }
         . . .
      });
   } catch (JwtException e) {
      e.printStackTrace();
      System.err.println("#### invalid token : " + token);
      throw new NotAuthorizedException(AUTHENTICATION_SCHEME + " realm=\"" + REALM + "\"");
   }
} else {
   System.err.println("#### Nessun token");
   throw new NotAuthorizedException(AUTHENTICATION_SCHEME + " realm=\"" + REALM + "\"");
}
```

Dopo aver verificato la validità del token, il server controlla che l'utente che ha effettuato la richiesta abbia i permessi necessari e, in caso negativo, risponde con il codice di errore HTTP 403 – non permesso.