UNIVERSITÀ DEGLI STUDI DI CATANIA

DIPARTIMENTO DI INGEGNERIA INFORMATICA

CORSO DI LAUREA IN INGEGNERIA INFORMATICA, SECONDO LIVELLO

Irene Baldacchino

Marco D'Alessandro

Just It

PROGETTO DI ADVANCED PROGRAMMING LANGUAGES

Anno Accademico 2020 – 2021

Indice

Capitolo 1 – Introduzione

JUST IT è un software implementato con lo scopo di gestire un meccanismo di prenotazione per consegne a domicilio, simile ad altri software già esistenti come Just Eat, Glovo o Deliveroo.

Dopo l'avvenuta registrazione o login, in base alla tipologia di utente (ristoratore o cliente), saranno visibili due diverse interfacce con funzionalità totalmente differenti al fine di simulare un vero e proprio meccanismo di scambio di ordini.

L'applicativo è stato suddiviso in quattro parti:

1. **Client**, implementato mediante l'utilizzo di C# poiché permette una programmazione ad eventi, essenziale per l'interazione dell'utente con l'interfaccia;
2. **Server**, implementato mediante l'utilizzo di Python insieme a Flask, framework web;
3. **Statistiche,** effettuate mediante l'utilizzo di R, nato per lo sviluppo di analisi statistica dei dati;
4. **Database** MySQL per rendere permanenti i dati inseriti.

Ogni capitolo sarà dedicato ad ogni singola parte implementativa del software, evidenziando alcune scelte effettuate durante la sua realizzazione.

Capitolo 2 – Client

Il client, quindi la parte dedicata all'interazione con l'utente, è stata implementata utilizzando come linguaggio di programmazione C#.

Questo ha permesso la creazione di un'interfaccia minimale per la generazione di eventi che eseguono chiamate REST al server.

Inizialmente, l'utente visualizzerà un'interfaccia comune sia al ristoratore che al cliente, dove sarà possibile effettuare il login o la registrazione.

Quindi la schermata principale che visualizzerà, sarà la seguente:

Figura 1: Login

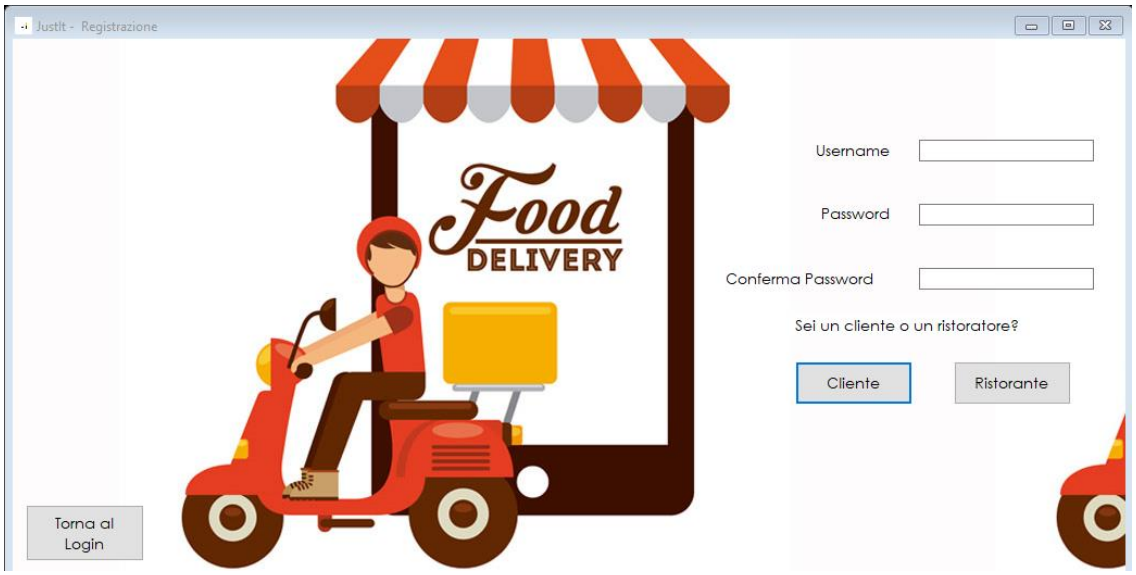


Figura 2: Registrazione

In questo caso inserendo le proprie credenziali sarà possibile accedere direttamente alla home associata al ristoratore o a quella del cliente.

2.1 – Ristoratore

Nel caso in cui l'utente risulti essere un ristoratore, verrà visualizzata la seguente homepage:

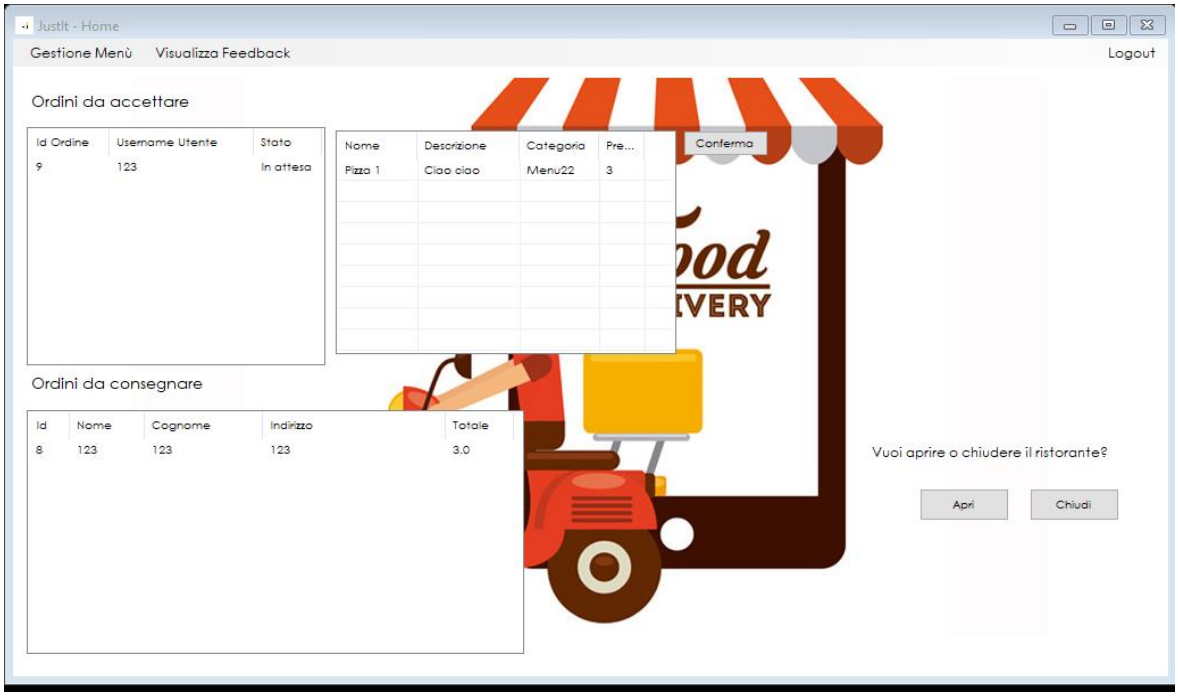


Figura 3: Home Ristorante

Da questa avrà la possibilità di:

1. **aprire/chiudere** il ristorante (in modo tale che il cliente possa visualizzarlo ed abbia la possibilità di poter effettuare un ordine);
2. **visualizzare** gli ordini effettuati dai clienti;
3. **cambiare lo stato dell'ordine in confermato o spedito;**
4. **inserire, visualizzare o cancellare (gestire)** i menù con i relativi piatti;

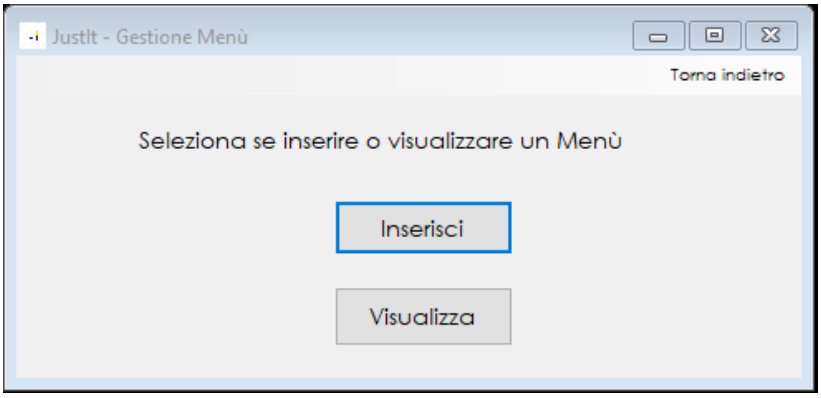


Figura 4: Gestione Menù

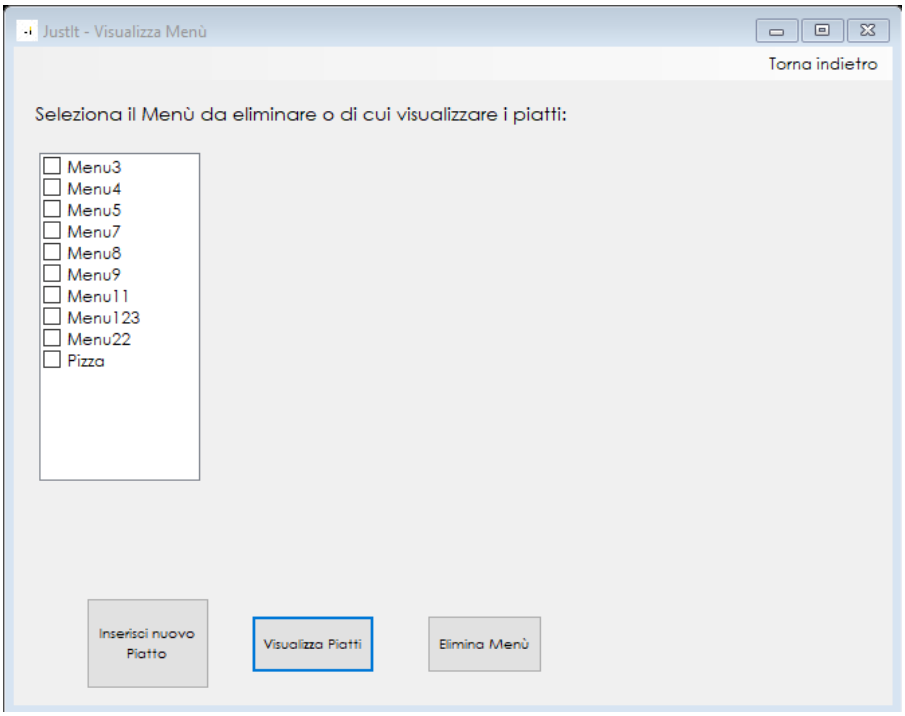


Figura 5: Visualizza/Elimina Menù

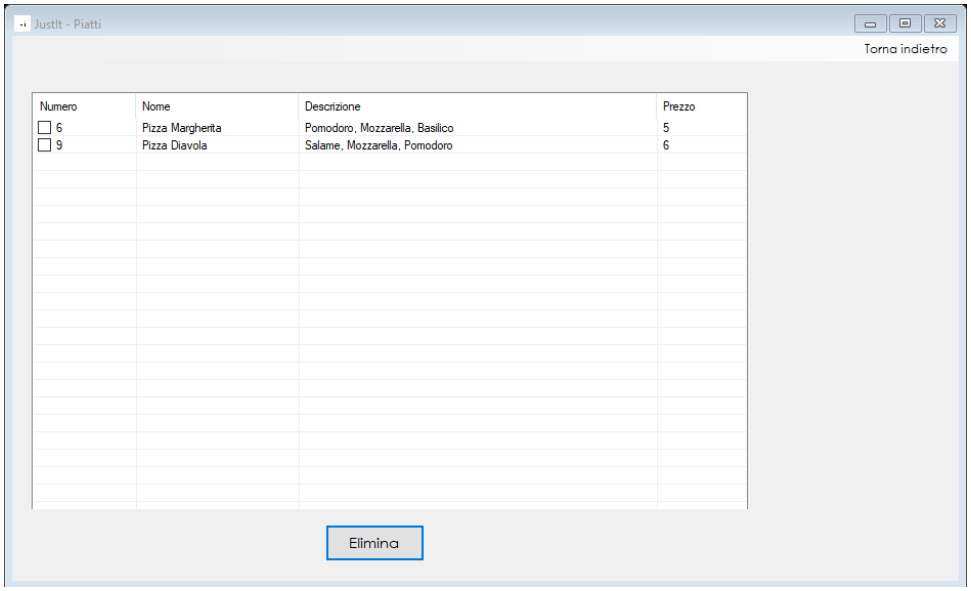


Figura 6: Visualizza Piatti

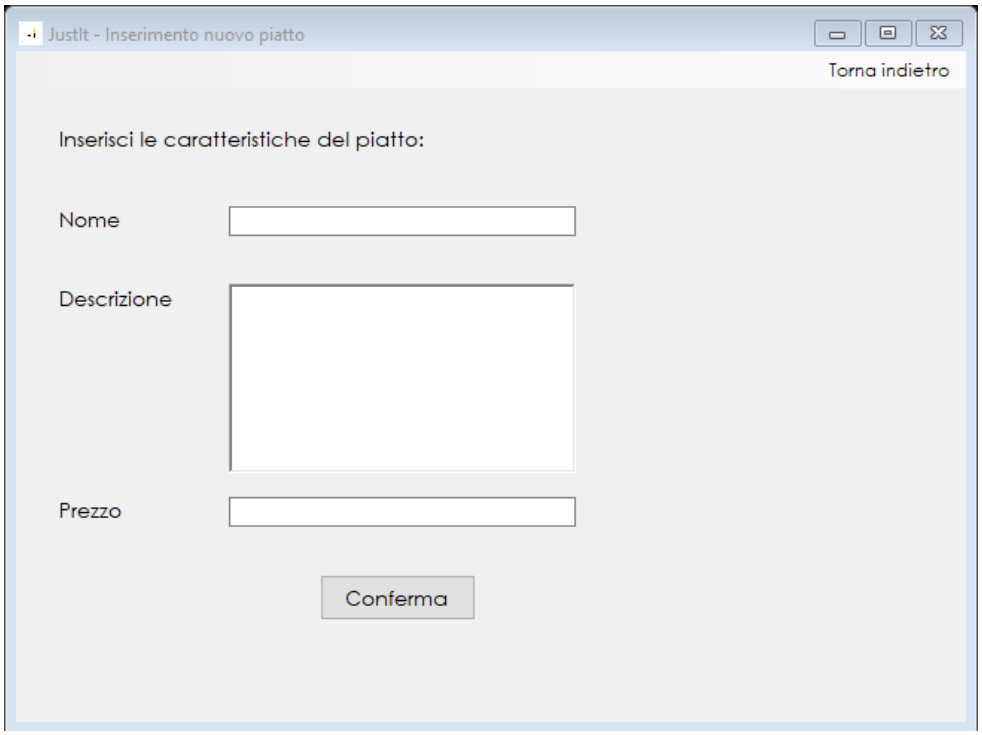


Figura 7: Inserisci nuovo piatto

5. **visualizzare il feedback** rilasciato dagli utenti.

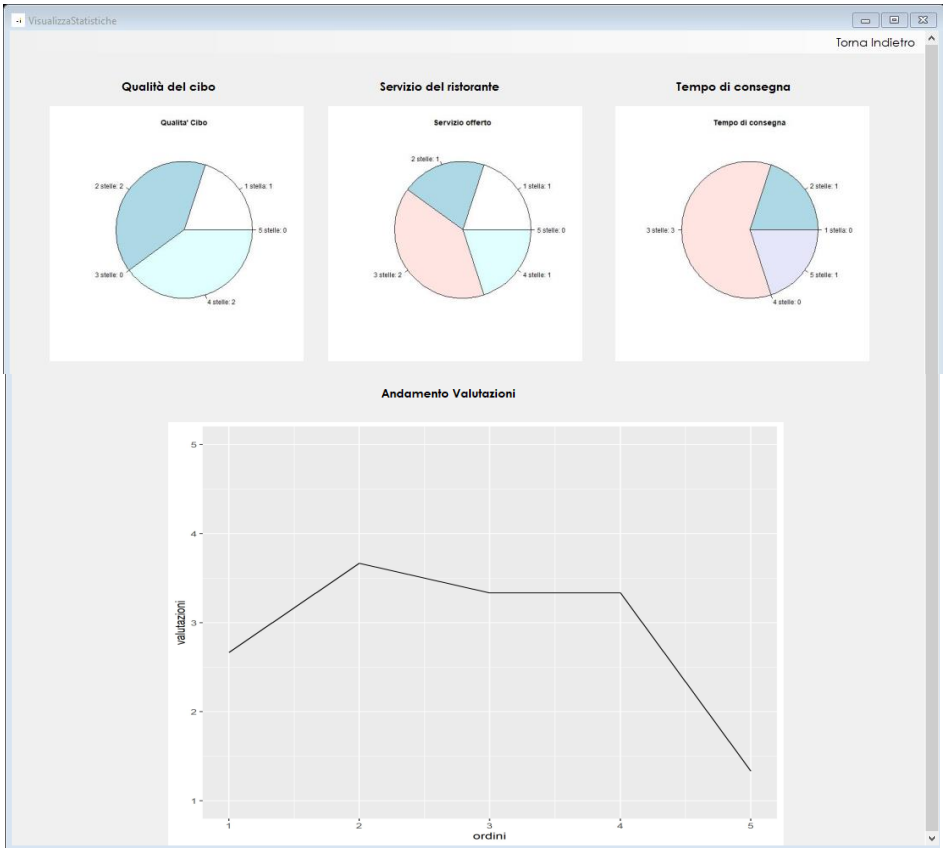


Figura 8: Visualizza Feedback

2.2 – Cliente

Nel caso in cui l'utente risulti essere un cliente, verrà visualizzata la seguente homepage:

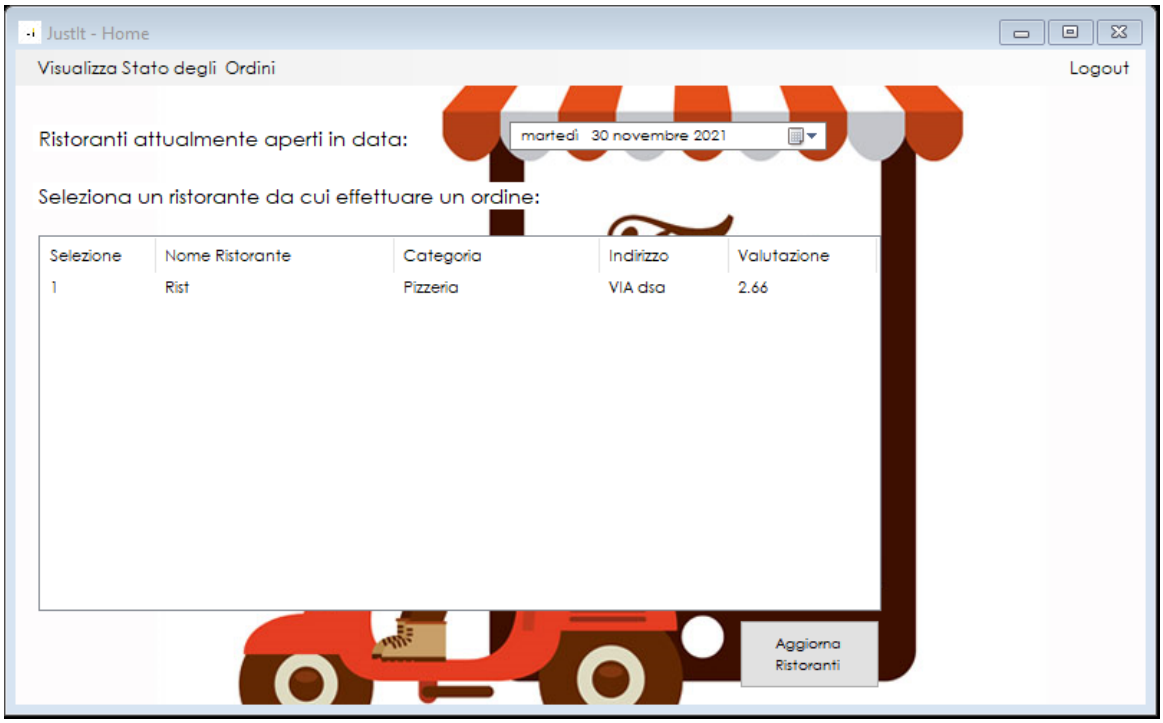


Figura 9: Home Cliente

Da questa avrà la possibilità di:

1. **Visualizzare i ristoranti aperti;**
2. **Visualizzare il menù** del ristorante scelto**;**

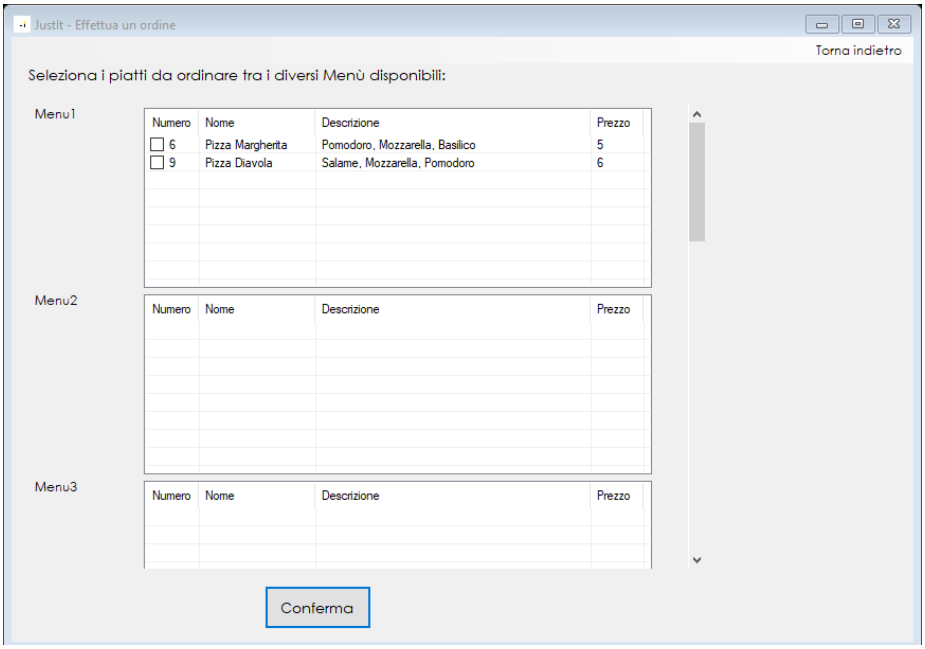


Figura 10: Visualizza menù di un ristorante scelto

3. **Effettuare un ordine;**
4. **Visualizzare lo stato degli ordini attuali e precedenti;**

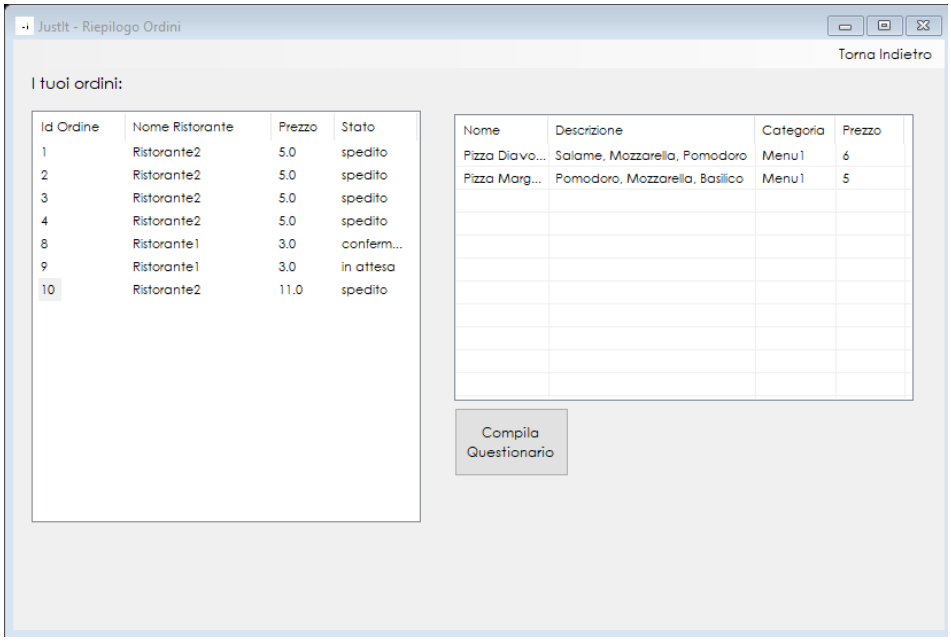


Figura 11: Visualizza riepilogo degli ordini

5. **Rilasciare un feedback al ristorante** dopo aver ricevuto un ordine.

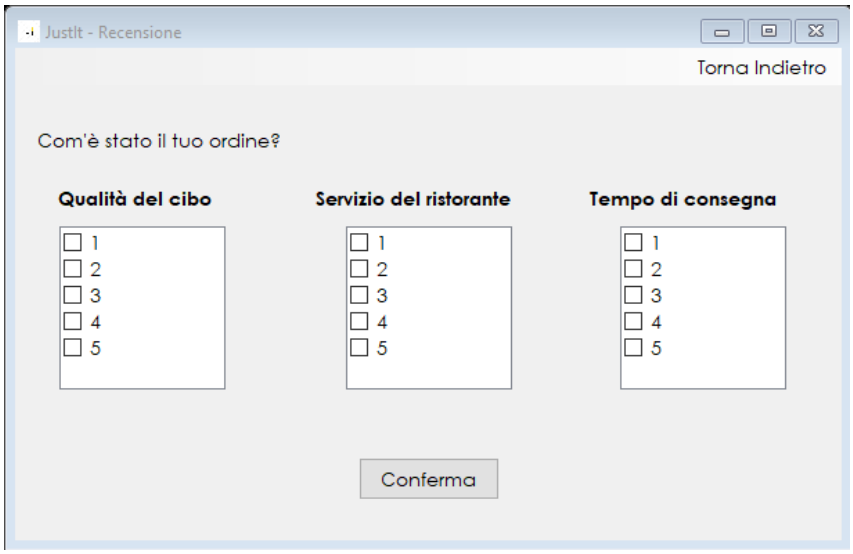


Figura 12: Inserisci Feedback

2.3 – Analisi di alcune scelte implementative

Al fine di effettuare un controllo continuo sullo stato degli ordini presenti nel database, è stato creato un **Thread,** il cui compito è quello di inserire gli aggiornamenti rilevati nella home del ristorante. Durante l'implementazione, una problematica rilevata è stata quella che il Thread non potesse modificare un oggetto creato dal Thread principale. Di conseguenza, è stato necessario introdurre l'utilizzo dei **delegati** e del metodo **Invoke()**.

```
public delegate void UpdateUI();

2 riferimenti
public void createThreadControlloOrdini()
{
    Thread t = new Thread(new ThreadStart(controlloOrdini));

    t.Start();
}
```

Figura 13: Creazione delegato e thread

```
1 riferimento
private async void controlloOrdini()
{
    while (controlloOrdiniInAttesa)
    {

        Dictionary<string, string> dicOrdine = new Dictionary<string, string>();

        dicOrdine.Add("mod", "in attesa");

        ordini = await functionRest.createRequest<ordine>(dicOrdine, "/ordine");

        listView1.Invoke(new UpdateUI(aggiornaListViewOrdiniAttesa));

        Dictionary<string, string> dicOrdine2 = new Dictionary<string, string>();

        dicOrdine2.Add("mod", "confermato");

        ordini_utenti = await functionRest.createRequest<ordine_utente>(dicOrdine2, "/ordine");

        listView2.Invoke(new UpdateUI(aggiornaListViewOrdiniSpedire));

        Thread.Sleep(5000);

    }
}
```

Figura 14: funzione controllo Ordini

Al fine di effettuare le diverse tipologie di richieste GET, PUT, DELETE e POST al server, sono state implementate molteplici funzioni.

In particolare, per le GET, ne sono state sviluppate due diverse:

1. *createRequest()* con lo scopo di effettuare una richiesta con dei parametri passati mediante un dizionario;
2. *createRequestWithoutParameters()* con lo scopo di effettuare una richiesta senza dei parametri.

```
//Metodo utilizzato per la creazione di una richiesta GET
17 riferimenti
public static async Task<List<T>> createRequest<T>(Dictionary<string, string> parameters, string path)
{
    var url = path + createUrlDictionary(parameters);

     List<T> result;

     result = await getAsync<T>(url);

     return result;

}

//Metodo utilizzato per la creazione di una richiesta GET senza parametri
1 riferimento
public static async Task<List<T>> createRequestWithoutParameters<T>(string path)
{
    List<T> result;

    result = await getAsync<T>(path);

    return result;
}
```

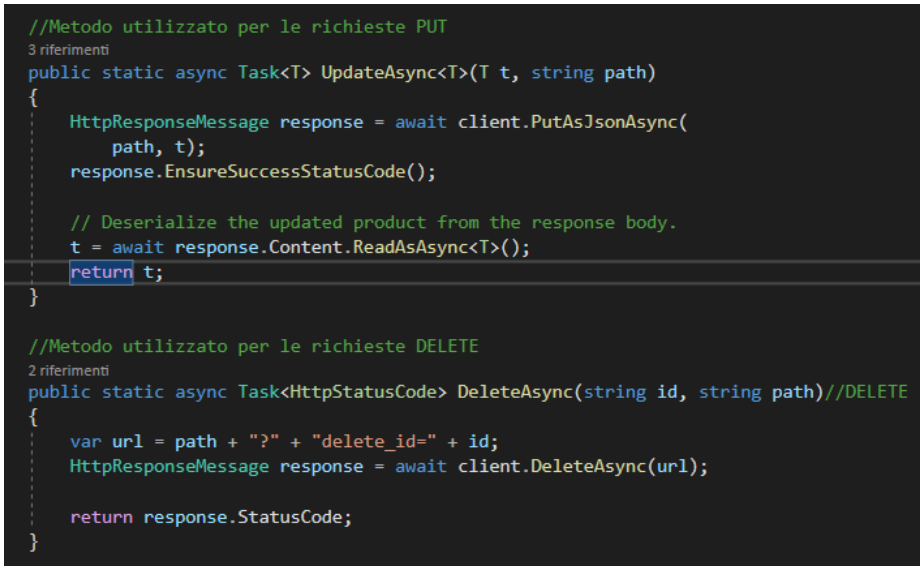
Figura 15: GET

Per effettuare delle richieste POST, sono state create, anche in questo caso, due funzioni:

1. *CreateAsync* con lo scopo di effettuare delle richieste delle quali viene restituito il solo status code;
2. *CreateAsyncReturnValue,* invece, oltre lo status code, riceve dei dati presenti nel body della risposta.

```
//Metodo utilizzato per le richiest POST con nessun valore di ritorno
5 riferimenti
public static async Task<Uri> CreateAsync<T>(T t, string path)
{

    HttpResponseMessage response = await client.PostAsJsonAsync(path, t);
    response.EnsureSuccessStatusCode();

    // return URI of the created resource.
    return response.Headers.Location;
}

//Metodo utilizzato per le richiest POST con un valore di ritorno
2 riferimenti
public static async Task<Dictionary<string,string>> CreateAsyncReturnValue<T>(T t, string path)
{
    HttpResponseMessage response = await client.PostAsJsonAsync(path, t);
    response.EnsureSuccessStatusCode();

    var repositories = await System.Text.Json.JsonSerializer.DeserializeAsync<Dictionary<string,string>>(await response.Content.ReadAsStreamAsync());
    return repositories;
}
```

Figura 16: POST



```
//Metodo utilizzato per le richieste PUT
3 riferimenti
public static async Task<T> UpdateAsync<T>(T t, string path)
{
    HttpResponseMessage response = await client.PutAsJsonAsync(
        path, t);
    response.EnsureSuccessStatusCode();

    // Deserialize the updated product from the response body.
    t = await response.Content.ReadAsAsync<T>();
    return t;
}

//Metodo utilizzato per le richieste DELETE
2 riferimenti
public static async Task<HttpStatusCode> DeleteAsync(string id, string path)//DELETE
{
    var url = path + "?" + "delete_id=" + id;
    HttpResponseMessage response = await client.DeleteAsync(url);

    return response.StatusCode;
}
```

Figura 17: PUT e DELETE

Capitolo 3 – Server

Il server permette di gestire la registrazione, il login e tutte le funzioni relative ai clienti e ai ristoratori. E' stato implementato in Python tramite l'ausilio del framework web "Flask", il quale permette al programmatore di gestire ciascuna funzione tramite l'utilizzo di opportune route, ciascuna delle quali richiamabile tramite uno specifico URL.

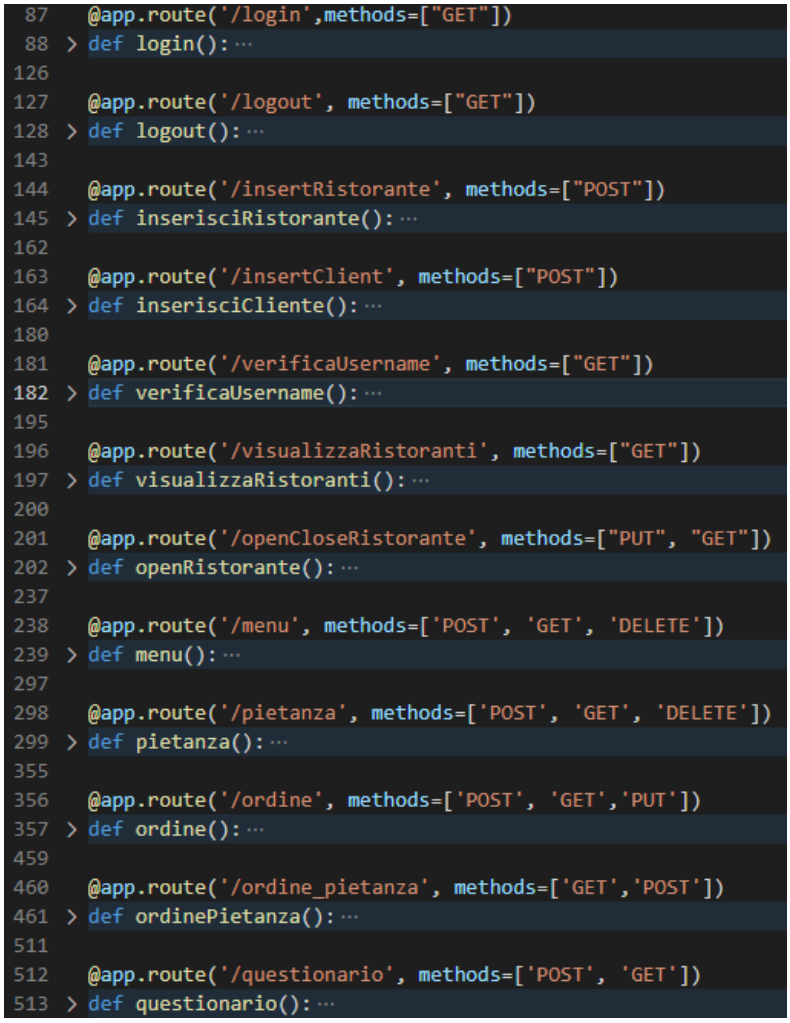


Figura 18: Panoramica Route

Ciascuna route può implementare una o più funzioni, ciascuna delle quali caratterizzata da un certo tipo di richiesta e/o dai parametri passati dal client.

Possono essere divise in tre gruppi, in relazione alle funzionalità e alle entità coinvolte:

- Route relative a fase di login e registrazione: ***/login, /logout, /insertCliente, /insertRistorante e /verificaUsername;***
- Route relative alla gestione delle funzionalità Cliente e Ristoratore: ***/OpenCloseRistorante, /menu, /pietanza, /ordine e /ordine_pietanza;***

- Route relative alle statistiche: ***/questionario.***

3.1 – Gestione Database

Oltre all'interazione client-server, viene anche gestita la connessione al Database MySQL per l'aggiunta, la rimozione, la modifica e la lettura dei dati relativi agli utenti e agli ordini. Per tale scopo è stata usata la libreria di Python "mysql-connector". Il funzionamento prevede una prima fase di connessione al database tramite la funzione *connectToDb( ),* la quale restituisce un MySQLConnection object che verrà usato per l'esecuzione delle varie query.

```
def connectToDb():
    cnx = mysql.connector.connect(user = userDB, password = pswUserDB, host = hostDB)
    mySQLcursor = cnx.cursor()
    try:
        mySQLcursor.execute("use {};".format(DB_NAME))
    except mysql.connector.Error as err:
        print("Il database non esiste, bisogna crearlo!")
        if err.errno == errorcode.ER_BAD_DB_ERROR:
            mySQLcursor.execute("use {};".format(DB_NAME))
    return cnx
```

Figura 19: connessione al Database

3.2 – Analisi di alcune scelte implementative

Al fine di garantire maggiore pulizia e leggibilità del codice è stato scelto di fornire più funzionalità a ciascuna route tramite la modifica dei parametri in ingresso piuttosto che associare a ciascuna funzionalità una route diversa.

La differenziazione può avvenire in un primo momento tramite il tipo di richiesta **(GET, POST, PUT, DELETE),** e nel caso in cui più funzionalità condividano lo stesso tipo, tramite uno switch-case. Un esempio si ha nella route relativa alle statistiche ordini. Si ha in particolare la possibilità di inserire dati nel database tramite una richiesta di tipo POST, ma anche di prelevare ciascuna statistica tramite richieste di tipo GET che differiscono per il valore del parametro "mod" passato in ingresso.

```python
@app.route('/questionario', methods=['POST', 'GET'])
def questionario():
    cnx = connectToDb()
    cursor = cnx.cursor()
    match request.method:
        case 'POST':  #Inserimento questionario
            data = request.json
            try:
                query = "insert into questionario(id_ordine,qualità_cibo,servizio_ristorante,tempo_consegna) values ({},{},{},{})".format(da
                cursor.execute(query)
                cursor.execute("update ordine set compilato = 1 where id = {}".format(data.get("id_ordine")))
                cnx.commit()
            except mysql.connector.Error as err:
                return jsonify(isError= True,
                    message= err.msg,
                    statusCode= 400,
                    )
            return jsonify(isError=False,
                           message="Success",
                           statusCode=200
                           )
```

```python
        case 'GET':
            mod = request.args.get('mod')
            try:
                cursor.execute("select qualità_cibo, servizio_ristorante, tempo_consegna from questionario join ordine on questionario.
                res = cursor.fetchall()
            except mysql.connector.Error:
                        return jsonify(isError= True,
                            message= "Errore richiesta",
                            statusCode= 400,
                            )
            qualita = []
            servizio = []
            tempo = []
            for row in res:
                    qualita.append(str(row[0]))
                    servizio.append(str(row[1]))
                    tempo.append(str(row[2]))

            match mod:
                case 'media_valutazioni':
                    cmd = [command, path_media] + qualita + servizio + tempo
                    x = subprocess.check_output(cmd, universal_newlines=True)
                    media = (x.split('"'))[1]
                    jsonResult = []
                    jsonResult.append({
                                "media": media[0:4]
                            })
                    return jsonify(jsonResult)

                case 'stats1':
                    cmd = [command, path_stats1] + qualita + servizio + tempo
                    x = subprocess.check_output(cmd, universal_newlines=True)
                    image=open(".\stats1.jpg","rb")
                    response=send_file(image,as_attachment=True, download_name='myfile.jpg')
                    return response

                case 'stats2':
                    cmd = [command, path_stats2] + qualita + servizio + tempo
                    x = subprocess.check_output(cmd, universal_newlines=True)
                    image=open(".\stats2.jpg","rb")
                    response=send_file(image,as_attachment=True, download_name='myfile.jpg')
                    return response
```

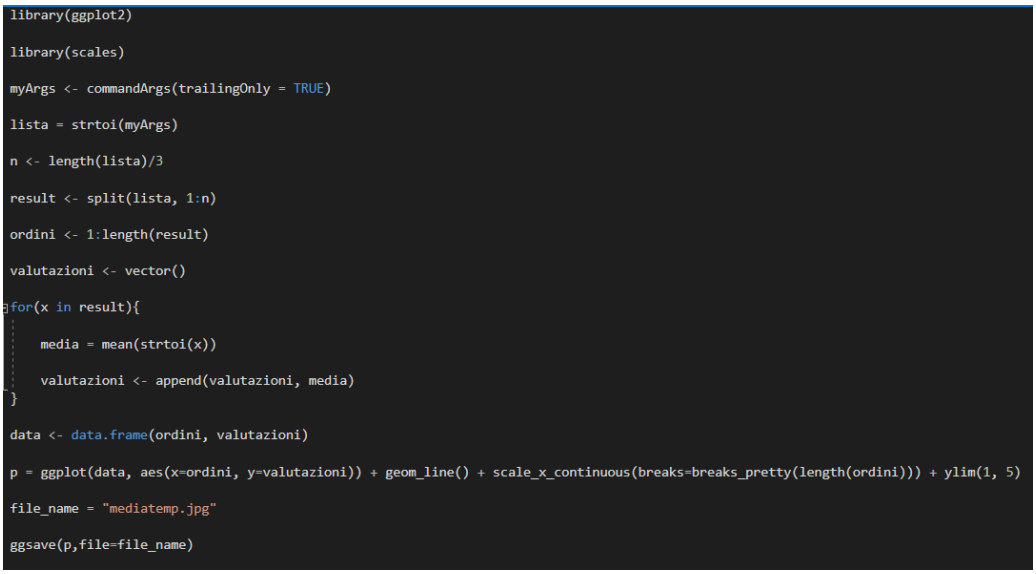
Figura 20: route questionario

Capitolo 4 – Script R

Il linguaggio utilizzato al fine di effettuare un'analisi statistica dei dati inseriti è R.

Sono stati implementati cinque scripts:

1. Calcolare la media delle valutazioni di ciascun ristorante;
2. Plottare un grafico a torta per rappresentare le valutazioni inerenti alla qualità del cibo;
3. Plottare un grafico a torta per rappresentare le valutazioni inerenti al servizio del ristorante;
4. Plottare un grafico a torta per rappresentare le valutazioni inerenti al tempo di consegna;
5. Plottare l'andamento delle valutazioni.



```
library(ggplot2)

library(scales)

myArgs <- commandArgs(trailingOnly = TRUE)

lista = strtoi(myArgs)

n <- length(lista)/3

result <- split(lista, 1:n)

ordini <- 1:length(result)

valutazioni <- vector()

for(x in result){

    media = mean(strtoi(x))

    valutazioni <- append(valutazioni, media)
}

data <- data.frame(ordini, valutazioni)

p = ggplot(data, aes(x=ordini, y=valutazioni)) + geom_line() + scale_x_continuous(breaks=breaks_pretty(length(ordini))) + ylim(1, 5)

file_name = "mediatemp.jpg"

ggsave(p,file=file_name)
```

Figura 21: Codice in R relativo al calcolo sull'andamento delle valutazioni

Per la realizzazione di questo script è stata utilizzata un tipo particolare di lista che serve per rappresentare un insieme di dati, cioè data.frame. Successivamente utilizzando la libreria **ggplot2** insieme alla frame creata, è stato possibile costruire l'apposito grafico.

Capitolo 5 – Conclusioni

Nel progetto realizzato si è cercato di utilizzare e sfruttare le diverse potenzialità di ogni singolo linguaggio, in particolare:

1. C# è stato scelto in quanto offre paradigmi di programmazione ad eventi, considerata ideale per la realizzazione e l'interazione con l'utente;
2. Python è stato scelto in quanto la sua sintassi risulta molto semplice, semplificando la leggibilità del codice e riducendo i costi di manutenzione del programma;

3. R è stato scelto in quanto ideale per lo sviluppo di calcoli statistici, mettendo a disposizione una vastità di metodi e la possibilità di creare grafici.

L'applicativo risulta essere una base che potrà essere integrata con ulteriori funzionalità in grado di ampliare le possibilità di utilizzo offerte agli utenti.