



Università
di Catania



Corso di Advanced Programming Languages a.a 2021/22

Vincenzo Pluchino 1000023070, Philip Tambè
1000015375

Relazione progetto: InstaFix

[Repository Github](#)

Sommario

1. Introduzione.....	2
2. Architettura dell'applicazione e linguaggi utilizzati	3
3. Requisiti e funzionalità implementate	4
4. Scelte implementative e aspetti salienti del codice.....	7

4.1 Client utente (C#)	7
4.2 Server (Go)	16
4.3 Client professionista (Python).....	24
5 Conclusioni e risultati	36

1. Introduzione

Lo scopo di questo elaborato è la realizzazione di un'applicazione denominata "*InstaFix*". L'obiettivo di questo applicativo client-server è quello di mettere in contatto l'utente con i professionisti della zona per risolvere i problemi quotidiani casalinghi: riparazioni, idraulica, pulizia, problemi di impianto elettrico, giardinaggio, falegnameria e tanti altri servizi che richiedono l'intervento di uno specialista.

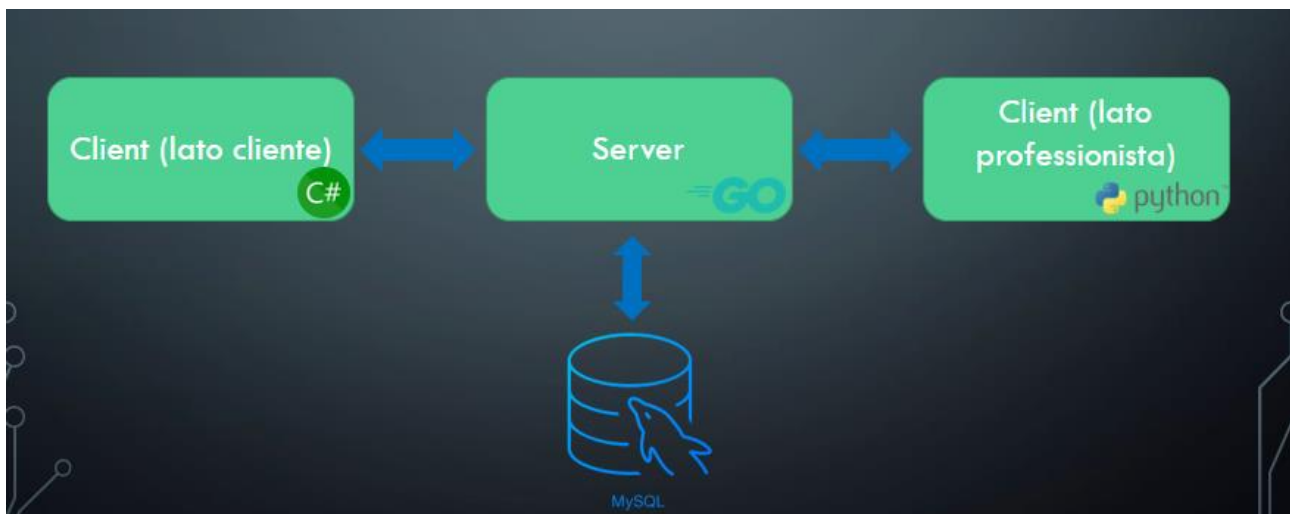
I client a disposizione in questo caso saranno due: uno verrà utilizzato dall'utente in questione, il fruitore principale, ovvero colui che ha bisogno di un intervento da parte di un esperto del settore; l'altro invece viene usufruito dal professionista, che gli consentirà di accettare/rifiutare proposte di lavoro e altre funzionalità che verranno descritte meglio nei paragrafi successivi. Entrambi, per poter utilizzare il software, hanno bisogno di registrarsi così da creare un account e utilizzare e-mail e password per poter effettuare il login e accedere al programma.

Le interazioni tra i due tipi di utente (cliente e specialista) verranno gestite dal server che si occuperà di inoltrare richieste e aggiornare le informazioni necessarie.



Logo dell'applicazione

2. Architettura dell'applicazione e linguaggi utilizzati



Architettura di riferimento di InstaFix

Come è possibile osservare dalla figura, l'elaborato consta di tre entità ognuna sviluppata con un linguaggio di programmazione differente.

Come già accennato nel capitolo introduttivo, il server funge da intermediario tra i due client ed è l'unica entità che ha accesso diretto al database; per ragioni di semplicità e per scelta progettuale, si è scelti di utilizzare un DB relazione di tipo mySQL in quanto si è ritenuto il più indicato per questo tipo di applicazione.

C# è stato utilizzato per sviluppare l'applicativo client dell'utente. La scelta è stata orientata verso questo linguaggio principalmente perché è il più adatto per la creazione di interfacce grafiche (GUI) intuitive e immediate, il che è ottimo essendo il client lato utente quello con più funzionalità disponibili. Oltre ad alcune librerie standard come, ad esempio, System.Net.Http che consente di effettuare e ricevere richieste http è stata utilizzata la libreria Netwonsoft (scaricabile sull'IDE

come pacchetto) che contiene una classe Json dotata di metodi per serializzare e deserializzare oggetti json.

Go è stato selezionato invece per lo sviluppo del server. In genere, è un buon compromesso come linguaggio per interfacciarsi con un DB in quanto ha delle librerie (come net/http o database/sql) che offrono delle funzioni in grado di eseguire query, generare risposte http in diversi formati, serializzare in json, ecc...

Una libreria esterna utilizzata in questo caso è <https://github.com/go-sql-driver/mysql> che si comporta come da driver vero e proprio per poter eseguire funzioni specificatamente per database di tipo MySQL (di fatto quindi non basta importare database/sql, è essenziale installare questo “driver”).

Python è il terzo linguaggio di InstaFix ad aggiungersi alla lista. È stato selezionato in quanto molto immediato e meno complesso nella struttura, ma soprattutto perché era ideale per poter sviluppare il client ad uso del professionista, che progettualmente ha bisogno di un’interfaccia grafica molto più minimale rispetto a quella del client lato cliente/utente ed ha anche meno funzioni da implementare. Le librerie maggiormente utilizzate sono state requests per l’inoltro di richieste dal client, mentre per l’interfaccia grafica si è preferito utilizzare tkinter. Inoltre, si è fatto uso sporadico di altre librerie come fpdf per la creazione delle fatture in PDF e PIL per la gestione del del logo di Instafix all’interno della GUI.

L’intero progetto è stato sviluppato utilizzando Visual Studio 2022 come IDE per il client in C#, GoLand di JetBrains per il server in Go e Visual Studio Code per il secondo client in Python.

3. Requisiti e funzionalità implementate

Di seguito verranno elencate, per i client e il server di InstaFix, le funzionalità disponibili e i requisiti principali su cui si basa il funzionamento del software nel suo insieme.

- **Client utente (lato cliente):**

- Registrazione di un account personale e accesso all'app tramite login (l'e-mail sarà univoca, non è possibile avere due account con lo stesso indirizzo di posta elettronica)
- Creazione di un ticket (ovvero una richiesta per il problema che l'utente vuole risolvere) ove viene specificato di che categoria si stia parlando (per maggiori dettagli consultare la fine di questo capitolo), un titolo e una descrizione del problema.
- Scelta di un professionista, a piacere, da parte dell'utente a cui verrà mostrato un elenco di specialisti indicizzati per recensione (verranno mostrati solo esperti del settore indicato nel ticket appena creato)
- Visualizzazione dei propri ticket in un'apposita sezione, dove è possibile consultare le informazioni principali e lo stato dei ticket (per maggiori informazioni sullo stato, vedere la fine di questo capitolo)
- Accettazione o rifiuto del preventivo, inviato dal professionista, riferito a uno dei ticket precedentemente creati.
- Download della fattura di una prestazione lavorativa eseguita da un professionista, in riferimento a un ticket a cui è stato accettato il preventivo fornito.
- Recensione dello specialista, che ha concluso il lavoro concordato, con un voto che va da 1 a 5

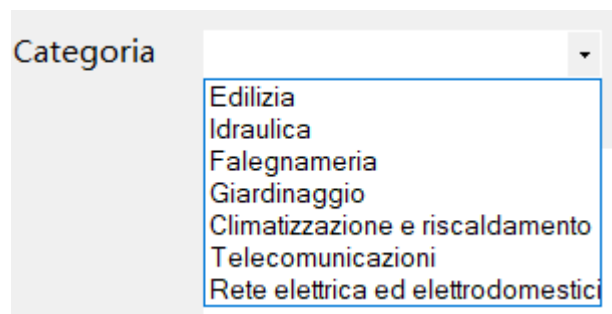
- **Client professionista**

- Registrazione account e accesso con login
- Accettazione o rifiuto di un ticket di richiesta di lavoro in cui lo specialista è stato scelto
- Visualizzazione dei ticket e delle loro info principali (compreso lo stato)
- Compilazione e invio di un preventivo relativo a un ticket accettato
- Upload della fattura a lavoro ultimato, che il cliente potrà consultare in qualsiasi momento

- **Server**

- Inserire, recuperare ed aggiornare i dati del db

- Gestire le interazioni tra i due client
- Fornire degli endpoint API per permettere ai client di effettuare richieste al server
- Gestire le richieste e inviare risposte a quest'ultime prelevando le informazioni necessarie dal database



Possibili categorie selezionabili durante la creazione di un ticket

Le categorie a cui un ticket può appartenere corrispondono di fatto ai settori di cui i professionisti sono specializzati: sono in totale sette quelli attualmente disponibili. Questo vuol dire che, ad esempio, se un professionista registrandosi indica come categoria di riferimento l'idraulica egli verrà mostrato ogni volta che un qualsiasi utente crei un ticket con categoria "Idraulica".

STATI TICKET

creato ---> ticket appena creato dall'utente e ha scelto il professionista, si è in attesa di risposta da parte del professionista
in attesa ---> il lavoratore ha inviato il preventivo, si attende la risposta dell'utente
in corso ---> l'utente ha accettato il preventivo, il professionista eseguirà il lavoro nella data e ora scelta
rifiutato ---> l'utente ha rifiutato il preventivo OPPURE il professionista ha rifiutato la richiesta del ticket
finito ---> il ticket è ufficialmente terminato quando il professionista invierà la fattura
votato ---> l'utente ha recensito il lavoratore relativamente al lavoro svolto per questo ticket

I vari stati che un ticket di InstaFix può assumere

Per poter avere consistenza e coerenza i ticket avranno un proprio stato, che descrive una particolare condizione in cui attualmente si trova la richiesta del cliente. Nell'immagine soprastante vengono indicati i sei possibili stati e come

avviene la transizione da uno stato a un altro: in questo modo sia utente che professionista possono “tracciare” gli aggiornamenti di un ticket così da poter intuire se devono svolgere un’azione o aspettare che la controparte effettui un’operazione.

(es. se il ticket si trova in stato di “in attesa”, il trigger che ha modificato lo stato è stato l’invio del preventivo; in questo caso tocca all’utente effettuare l’azione di accettare o rifiutare il preventivo per far transitare il ticket in uno stato successivo)

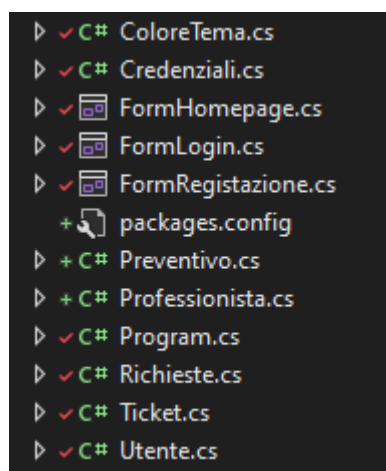
4. Scelte implementative e aspetti salienti del codice

Questo capitolo, diviso in tre sezioni, ha lo scopo di scendere più nel dettaglio su come sono state sviluppate le funzionalità sopra descritte e dare risalto alle righe di codice più rilevanti in termini di complessità o importanza.

4.1 Client utente (C#)

Per lo sviluppo del client lato utente sono stati utilizzati principalmente i Windows Forms per costruire varie finestre e rendere la GUI più intuitiva possibile.

Oltre ai forms, sono diverse le classi utili che sono state create per la corretta esecuzione del software.



Alcune classi come utente, professionista, preventivo ecc replicano all’incirca le

tabelle del database, così da poter creare oggetti di quella classe qualora servino. Questi file, infatti, contengono solamente le proprietà della classe (nome, cognome, indirizzo...) così da sfruttarne le potenzialità del linguaggio. Program.cs è il punto di partenza del programma, in cui in pochissime righe si specifica quale Form rendere visibile all'avvio.

```
public async Task<string> HttpPostAsync(string uri, HttpContent cont)
{
    string content = null;
    var client = new HttpClient();
    var response = await client.PostAsync(uri, cont);
    if (response.IsSuccessStatusCode)
    {
        content = await response.Content.ReadAsStringAsync();
        Debug.WriteLine(content);
    }
    return content;
}
```

Il metodo più utilizzato della classe Richieste

Di particolare importanza è la classe Richieste, il quale contiene al suo interno i metodi per poter effettuare richieste web al server sfruttando la classe httpClient: Tramite l'uso di async e await è possibile effettuare richieste al server in modo asincrono, facendo quindi uso della programmazione asincrona di C# basato su **TAP** (pattern asincrono basato su Task). In questo modo è possibile avere un flusso di esecuzione non bloccante, che eviti all'utente di aspettare obbligatoriamente la risposta del server prima di procedere con un'altra attività. Il primo Form che compare all'avvio del software è quello che si occupa del login: verranno sorvolati, sia per questo che per tutti gli altri form, gli aspetti grafici e il codice autogenerato durante la creazione di bottoni, aree di testo e quant'altro... La parte più rilevante, che verrà mostrata solo qui in quanto simile a tutte le altre web request presenti nel client, riguarda la connessione tramite metodo post a un endpoint API del server (locale in questo caso...)


```

try
{
    var datiUtente = new Dictionary<string, string>
    {
        {"email", email },
        {"password", pass }
    };

    var datiDaInviare = new FormUrlEncodedContent(datiUtente);
    var result = await loginPost.HttpPostAsync("http://localhost:8000/login", datiDaInviare);

    if (result.Equals("Credenziali errati"))
    {
        MessageBox.Show("L'email o la password sono errati. Ricontrolla i dati",
            "Login error", MessageBoxButtons.OK, MessageBoxIcon.Error);
    }
    else if (result.Equals("Errore generico"))
    {
        MessageBox.Show("Qualcosa è andato storto", "Errore"
        else
        {
            idUtente = result;
            new FormHomepage().Show();
            this.Hide();
        }
    }
}
}

```

13 riferimenti

```

public partial class FormLogin : Form
{
    public static string idUtente;
}

```

Metodo che gestisce l'evento click sul bottone "login"

FormUrlEncodedContent codifica il dizionario creato con i dati inseriti dall'utente nelle apposite caselle di testo utilizzando il tipo MIME `application/x-www-form-urlencoded`; si richiama poi il metodo `post` dell'istanza `loginPost` che è un oggetto della classe `Richieste` inserendo l'url di riferimento e il body del POST. In base alla risposta mandata dal server verrà mostrato un popup diverso.

Se il login va a buon fine, l'id dell'utente viene salvato in una variabile *statica* pubblica che verrà sfruttata per tutte le future richieste di quello stesso utente; questo meccanismo è stato implementato in quanto non si utilizzano i socket per la comunicazione client-server e quindi non c'è un concetto di sessione tra le due entità comunicanti.

La finestra relativa alla registrazione è di facile intuizione, consiste semplicemente nel controllo dei vari campi che l'utente deve inserire e l'invio dei dati tramite `PostAsync` al server per avviare la query relativa alla creazione dell'account.

N.B. tutte le request effettuate al server tramite API sono definite dentro un blocco `try catch`, così da gestire l'eccezione nel momento in cui il server sia offline o non disponibile al momento del tentativo di connessione da parte del client

```
// controlli formattazione email e psw
Regex emailCorretta = new Regex(@"^[w!#$%&'*\+\/=?\^_`{|}~]+(\.[w!#$%&'*\+\/=?\^_`{|}~]+)*" + @"
+ @((((\w)+\.)+[a-zA-Z]{2,4})|(([\0-9]{1,3}\.){3}[\0-9]{1,3}))$");
Match controlloEmail = emailCorretta.Match(email);
Regex pswCorretta = new Regex(@"^(?!.* )(?=.*[A-Z])(?=.*[a-z])(?=.*[0-9])(?=.*[.#!@$%^&*~]).{8,15}$");
Match controlloPassword = pswCorretta.Match(pass);
```

Utilizzo della classe *Regex* per il controllo della formattazione di e-mail e password. Questa classe, derivata dalla libreria *Text.RegularExpressions*, viene utilizzata per definire un pattern tramite una sequenza di caratteri così da cercare stringhe o file che corrispondono al pattern.

Dopo che l'utente si autentica viene reindirizzato nell'homepage che consiste in una finestra contenente diversi pulsanti che permettono di accedere alle varie funzioni della piattaforma.

A livello grafico, il colore del form cambierà ad ogni click in uno dei diversi pulsanti:

```
private void bottoneAttivo(object bottone)
{
    if (bottone != null)
    {
        if (bottoneCorrente != (Button)bottone)
        {
            bottoneDisattivo();
            Color colore = SelezionaTemaColore();
            bottoneCorrente = (Button)bottone;
            bottoneCorrente.BackColor = colore;
            bottoneCorrente.ForeColor = Color.White;
            bottoneCorrente.Font = new System.Drawing.Font("Microsoft Sans Serif", 10.00F,
            panelTitle.BackColor = colore;
            buttonBackHome.Visible = true;
        }
    }
}
```

Metodo (contenuto in *FormHomepage.cs*) per evidenziare il bottone cliccato e cambiare colore in modo random al pannello del form

Il metodo *SelezionaTemaColore* sceglie in modo casuale un colore da impostare come tema per la tutta la finestra dell'home e si avvale della classe *ColoreTema* che contiene una lista di molteplici colori espressi in formato esadecimale. *BottoneDisattivo* invece rende gli altri pulsanti meno in evidenza lasciando il colore standard bordeaux (solo il bottone cliccato cambierà colore).

```
// metodo per aprire altri form all'interno del pannello dove si
7 riferimenti
private void OpenForms(Form formFiglio, object bottone)
{
    if (formAttivo != null)
    {
        formAttivo.Close();
    }
    bottoneAttivo(bottone);
    formAttivo = formFiglio;
    formFiglio.TopLevel = false;
    formFiglio.FormBorderStyle= FormBorderStyle.None;
    formFiglio.Dock= DockStyle.Fill;
    this.panelSottostante.Controls.Add(formFiglio);
    this.panelSottostante.Tag = formFiglio;
    formFiglio.BringToFront();
    formFiglio.Show();
}
```

Metodo all'interno di FormHomepage.cs che consente di visualizzare un nuovo form all'interno della stessa finestra della homepage

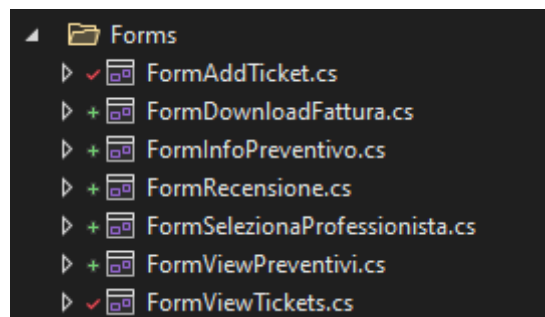
Ad ogni pulsante corrisponde una nuova finestra che avrà diversi elementi a seconda del tipo di funzione che deve svolgere. Il metodo *OpenForms* ha l'obiettivo di aprire un nuovo form dentro il form generale dell'homepage così da avere sempre il menù a portata di mano e nel frattempo interagire col servizio scelto; per farlo vengono utilizzate alcune proprietà generali dei forms come Dock, TopLevel, etc e il form viene aperto nell'elemento chiamato panelSottostante (è il pannello bianco a destra subito visibile non appena si apre la home).

```
private void buttonViewTicket_Click(object sender, EventArgs e)
{
    if (Forms.FormAddTicket.selectProfessionista == false)
    {
        addTicketBool = false;
        viewPreventiviBool = false;
        OpenForms(new Forms.FormViewTickets(), sender);
        labelTitle.Text = "I TUOI TICKET";
    }
    else
    {
        MessageBox.Show("Per poter proseguire è necessario selezionare un professionista per il ticket appena creato", "Attenzione!", MessageBoxButtons.OK);
    }
}
```

Metodo per gestire l'evento di click su uno dei pulsanti dell'homepage (in questo caso il pulsante "vedi i tuoi ticket")

I metodi per gestire gli eventi click sono pressochè simili: si impostano alcune variabili di controllo booleane true o false e si richiama la funzione *OpenForms* i cui parametri sono il nuovo form da visualizzare e la variabile *sender*, che è un riferimento all'oggetto/controllo che ha scatenato l'evento (in questo caso particolare è il bottone).

Il controllo sulla variabile statica booleana *selectProfessionista* viene fatto in modo tale che l'utente, dopo aver creato il ticket, deve subito selezionare il professionista a cui si vuole rivolgere e quindi non può interagire con altri servizi per evitare che il ticket rimanga incompleto (vedi messaggio di errore dello statement else nell'immagine sopra).



Elenco di tutti i form "secondari" apribili nella finestra Home

I vari form elencati nell'immagine costituiscono le implementazioni dei requisiti

riportati nel capitolo 3. **Requisiti e funzionalità implementate.** Il focus riguardo lo sviluppo di questi elementi verrà dato solamente ad alcuni forms, in quanto molti aspetti sono pressochè identici e risulterebbe ridondante descriverli.

Una parte interessante del codice è sicuramente quella che riguarda la funzionalità di poter visualizzare i preventivi.

```
Richieste getPreventivi = new Richieste();

try
{
    var idUser = new Dictionary<string, string>
    {
        {"id_utente", FormLogin.idUtente}
    };
    var datiDaInviare = new FormUrlEncodedContent(idUser);
    var result = await getPreventivi.HttpPostAsync("http://localhost:8000/getpreventivi", datiDaInviare);
    List<Preventivo> arrayPreventivi = JsonConvert.DeserializeObject<List<Preventivo>>(result);

    if (arrayPreventivi != null)
    {
        foreach (var item in arrayPreventivi)
        {
            listViewPreventivi.Items.Add(new ListViewItem(new String[] { item.IdTicket.ToString(), item.Costo.ToS
        }
    }
}
```

Parte del metodo "loadPreventivi" del file FormViewPreventivi.cs

La risposta che viene ritornata dal server utilizzando la route /getpreventivi è un oggetto json contenente uno o più preventivi. Per questo motivo, grazie alla libreria

Newtonsoft accennata nel capitolo 2. **Architettura**

dell'applicazione e linguaggi utilizzati, si è in grado di

deserializzare l'oggetto in una lista di oggetti di classe Preventivo; con un ciclo for si va poi ad aggiungere in una *ListView* (oggetto grafico di Windows Forms simile a una tabella) gli attributi che si desidera far visualizzare all'utente (come ad esempio costo, descrizione, data & ora...)

```

private void listViewPreventivi_MouseClick(object sender, MouseEventArgs e)
{
    for (int i = 0; i < listViewPreventivi.Items.Count; i++)
    {
        idPreventivo = listViewPreventivi.SelectedItems[0].SubItems[4].Text;
        var rectangle = listViewPreventivi.GetItemRect(i);
        if (rectangle.Contains(e.Location))
        {
            preventivoInfo = true;
            this.Hide();
        }
    }
}

```

Gestione dell'evento click sulla lista dei preventivi

Selezionando uno dei preventivi, la variabile booleana *preventivoInfo* passerà a true e tramite verifica nel codice dell'homepage che quest'ultima variabile abbia cambiato valore verrà aperto un altro form, *FormInfoPreventivo*, che consiste in una finestra in cui verranno stampati tutti i dettagli più specifici del preventivo cliccato; sarà inoltre possibile accettare o rifiutare il preventivo mentre si è in questa nuova finestra.

Per comprendere quale elemento è stato cliccato dall'utente, si utilizza l'attributo *Location* della classe *Event* che permette di ottenere la posizione del mouse nel momento in cui è stato generato l'evento; dopodiché col metodo *GetItemRect* si ottiene il preciso rettangolo in cui è stato applicato il controllo (il click in questo caso) e si preleva l'id del preventivo con l'attributo *SelectedItems*.

Prima di concludere questo paragrafo e passare alla descrizione del server, ci si soffermerà sulla gestione del Form che si occupa di far scaricare le fatture caricate dai professionisti a lavoro ultimato.

```

string pathSave;
if (RuntimeInformation.IsOSPlatform(OSPlatform.Linux))
{
    pathSave = Environment.GetFolderPath(Environment.SpecialFolder.UserProfile);
}
else
{
    // di default l'applicazione salva le fatture nel desktop
    pathSave = Environment.GetFolderPath(Environment.SpecialFolder.Desktop);
}

using (WebClient wc = new WebClient())
{
    wc.DownloadProgressChanged += wc_DownloadProgressChanged;
    wc.DownloadFileAsync(
        // Param1 = Link of file
        //
        new System.Uri(linkFattura),
        // Param2 = Path to save
        pathSave + "/fattura" + id_ticket + ".pdf"
    );
}

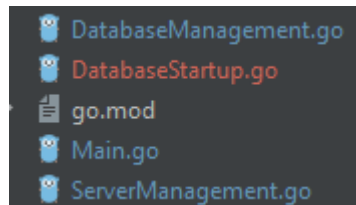
```

Il metodo downloadFattura del file FormDownloadFattura.cs; i parametri utilizzati per richiamarlo sono linkFattura e idTicket

La variabile *pathSave* definisce il percorso in cui verrà salvato il pdf della fattura: esso cambia a seconda del tipo di SO su cui sta girando il client (Windows o Linux). Successivamente, viene creata un'istanza della classe *WebClient* così da chiamare il metodo *DownloadFileAsync* che effettua il download senza bloccare il thread corrente. I parametri sono due: il secondo è la variabile *pathSave*, il primo è l'uri in cui si trova la risorsa; *linkFattura*, ovvero l'uri, per ragioni di sicurezza non è conosciuto a priori dal client: questo perché i file delle fatture si trovano in una directory del file system del server. Per ricavarlo quindi si effettua una richiesta al server che restituisce il percorso del file richiesto dall'utente, effettuando i dovuti controlli sul DB.

4.2 Server (Go)

Il server, a differenza dei due client, non presenta nessun tipo di interfaccia grafica: l'unica parte a cui l'amministratore potrebbe far attenzione è la console dove vengono stampati dei messaggi informativi o messaggi di errore se qualcosa dovesse andare storto.



Lo sviluppo del server consta nei quattro file visualizzabili nell'immagine; non avendo Go un concetto di classe vero e proprio, si è scelto come prassi di buona programmazione di evitare di accorpare tutto dentro la funzione main e di utilizzare altri due file per la gestione del server e del DB.

Tutto ciò ha portato alla creazione di un main molto snello:

```
var sqlDB *sql.DB

func main() {

    //DB
    sqlDB = startDB()

    fmt.Println("E' il primo avvio del database? Se si e hai bisogno di creare le tabelle inserisci 1  
Altrimenti, inserisci qualsiasi altro tasto per continuare e avviare il server: ")
    var answer string
    fmt.Scanf(" %s", &answer)
    strYes := "1"
    if answer == strYes {
        databaseSetup(sqlDB)
    }
    esempioQuery(sqlDB)

    // server
    startServer()
}
```

Funzione main del file main.go: è il punto di avvio del server

La variabile globale *sqlDB* è un puntatore che permette a qualsiasi altra funzione del programma di avere un riferimento del database, la cui connessione avviene

tramite la funzione *startDB()*.

Qualora fosse il primo avvio in assoluto ed è quindi necessario creare le tabelle, inserendo l'apposito numero verrà eseguita la funzione *databaseSetup* che è l'unica funzione del file *DatabaseStartup.go*. La funzione consiste semplicemente nell'esecuzione di tutte le query di tipo "CREATE TABLE".

Il file *ServerManagement.go* contiene tutte le funzioni necessarie per creare gli endpoint API, gestire le richieste prelevando le variabili del request body e generare le risposte da inviare ai client.

```
http.HandleFunc( pattern: "/getinfopreventivo", getinfopreventivo)
http.HandleFunc( pattern: "/acceptpreventivo", acceptpreventivo)
http.HandleFunc( pattern: "/denypreventivo", denypreventivo)
http.HandleFunc( pattern: "/downloadfattura", downloadfattura)
http.HandleFunc( pattern: "/getprofessionistidavotare", getprofessionistidavotare)
http.HandleFunc( pattern: "/voteprofessionista", voteprofessionista)
http.ListenAndServe( addr: ":8000", handler: nil)
```

Alcune definizioni delle routes definite all'interno della funzione startServer del file ServerManagement.go. E' una delle poche funzioni richiamate direttamente nel main ad avvio programma

Tramite l'importazione della libreria *net/http* è possibile registrare le route con la funzione *HandleFunc*: il primo parametro indica l'endpoint ovvero l'url che il client deve raggiungere per poter effettuare una richiesta; il secondo parametro indica la funzione che si occupa di gestire la richiesta ricevuta. *ListenAndServe* ha lo scopo di specificare in quale porta dell'host il server deve mettersi in ascolto.

La costruzione della risposta da mandare al client dipende dal tipo di risposta che si deve includere nella response http

```
func newticket(w http.ResponseWriter, req *http.Request) {
    req.ParseForm()
    title := req.Form.Get(key: "titolo")
    category := req.Form.Get(key: "categoria")
    description := req.Form.Get(key: "descrizione")
    email := req.Form.Get(key: "email")
    resp := newticketQuery(sqlDB, title, category, description, email)
    fmt.Println(resp)
    fmt.Fprintf(w, resp)
}
```

Tipica funzione di gestione di una request a una route del server

Il metodo *ParseForm* si occupa di leggere il response body delle Request effettuando il parse e inserendo il risultato nell'attributo *Form* di *req* (che è un *http.Request*); tramite la funzione *Get* si ottiene il valore associato alla chiave: è di fondamentale importanza che la key sia esattamente uguale carattere per carattere a quella inserita lato client, in caso contrario non si riuscirà a risalire al valore.

Una volta ottenuti tutti i valori si delega la gestione alle funzioni contenute nel file *DatabaseManagement.go* che si occupano di effettuare le query sul database. Nel caso più semplice, come quello nell'immagine sopra che ritrae la funzione *newticket*, la funzione che effettua la query ritorna una semplice stringa: è sufficiente quindi utilizzare *Fprintf* specificando come primo parametro una variabile di tipo *http.ResponseWriter* per poter generare la risposta al client.

N.B. Attenzione al Content Type della richiesta del client! Se esso non è di tipo *x-www-form-urlencoded* il metodo *ParseForm* non legge il body della richiesta e l'attributo *Form* sarà inizializzato a *nil* (il corrispettivo di null in Go). Ecco perché a pagina 9 viene specificato che il client utente effettua le richieste HTTP al server utilizzando questa determinata codifica

```
func getprofessionisti(w http.ResponseWriter, req *http.Request) {
    req.ParseForm()
    categoria := req.Form.Get(key: "categoria")
    resp := getProfessionistiQuery(sqlDB, categoria)
    fmt.Println(resp)
    w.Header().Set(key: "Content-Type", value: "application/json")
    w.WriteHeader(http.StatusCreated)
    json.NewEncoder(w).Encode(resp)
}
```

Funzione di gestione della route /getprofessionisti. Notare la differenza nella costruzione della response rispetto all'esempio precedente

In questo scenario, invece, la funzione *getProfessionistiQuery* dopo aver completato le query sql ritorna un array di “oggetti” di tipo Professionisti: la semplice *Fprintf* utilizzata prima non va più bene.

La soluzione è quella di utilizzare i metodi forniti dalla variabile *w* di tipo *ResponseWriter* come *Header().Set* e *WriteHeader* che permettono di impostare manualmente l'header della response HTTP. In questo caso si opta per avere un Content type del body di tipo json, rendendo quindi necessario utilizzare la libreria “encoding/json” per codificare l'array restituito dalla funzione (la variabile *resp*) in json.

L'ultimo file del server è DatabaseManagement.go, in cui sono presenti le funzioni richiamate dai gestori delle API request per interrogare il database di InstaFix.

```
// i nomi dei campi degli struct sono indicati con la lettera iniziale maiuscola per alcuni problemi
// derivati dall'incapsulamento dello struct stesso in json
type Ticket struct {
    Id          int    // `json:"id"`
    Stato       string // `json:"stato"`
    Categoria   string // `json:"categoria"`
    Titolo      string // `json:"titolo"`
    Descrizione string // `json:"descrizione"`
}

type Professionisti struct {
    Id          int
    Nome        string
    Cognome     string
    Professione string
    Recensione  float32
    Citta       string
}

type Preventivi struct {
    Id          int
    IdTicket    int
    IdProfessionista int
    Descrizione string
    MaterialiRicambi string
    Costo       float32
}
```

Definizione degli struct all'inizio del file DatabaseManagement.go

Una delle potenzialità di Go che sono state utilizzate per lo sviluppo del server sono le struct, con le quali è possibile definire dei tipi di dato da poter utilizzare come se fossero degli “oggetti” (è importante ricordare che non esiste il concetto di classe vero e proprio in Go). Queste strutture, che corrispondono ad alcune tabelle del DB, sono state create perché alcune response del server devono ritornare un elenco di uno o più elementi di un certo tipo (un elenco di ticket, di professionisti e via di seguito...). Ogni elemento, infatti, ha una serie di variabili (un po' come gli attributi) i cui tipi differiscono: ecco perché non era possibile risolvere la cosa con un semplice array o una lista.

```
func startDB() *sql.DB {
    db, err := sql.Open( driverName: "mysql", dataSourceName: "root:@tcp(localhost:3306)/instafix")
    //defer db.Close()

    if err != nil {
        fmt.Println(a...: "Controllare la funzione startDB del file DatabaseManagement.go per es
        " di aver inserito il corretto host/portass, id/password e il nome del dataabase")
        log.Fatal(err)
    }

    var version string

    err2 := db.QueryRow( query: "SELECT VERSION()").Scan(&version)
```

User:Psw

Host dove il servizio mySql è in run: porta

Nome del database

Funzione per avviare il collegamento con sql e il database. Si ricordi che è una delle funzioni richiamate dal main del programma

Con la libreria database/sql e il relativo drive disponibile su Github (vedi pagina 4) si è in grado di connettersi al database e al servizio di SQL tramite la funzione *Open* che ritorna una variabile puntatore di tipo **sql.DB*. È importante fare attenzione al secondo parametro, *dataSourceName*, in quanto sono contenuti tutti i parametri necessari che sono evidenziati ed esplicitati in rosso nell'immagine sopra; il limite sta nel fatto che il database deve essere già creato e si deve conoscere il nome del suddetto (nell'esempio dell'immagine ha il nome di "instafix").

```
func loginQuery(datab *sql.DB, email string, psw string) string {
    query := fmt.Sprintf( format: "SELECT email FROM credenziali WHERE email= '%s' AND psw='%s'", email, psw)
    result1, err1 := datab.Query(query)

    if result1.Next() == false {
        return "Credenziali errati"
    }
    if err1 != nil {
        fmt.Println(err1)
        return "Errore generico"
    }
    defer result1.Close()

    return "Credenziali corrette"
```

Funzione richiamata dall'handler della route /login

Una tipica funzione di *DatabaseManagement.go* consiste nell'eseguire una o più query costruendola con il metodo *Sprintf* così da utilizzare i parametri della funzione stessa per l'interrogazione (mail e password, ad esempio, per controllare se esistano nel DB e acconsentire il login). Utilizzando poi il puntatore al db si richiama la funzione *Query* ed essa viene eseguita, ritornando due elementi: il risultato dell'interrogazione e un eventuale errore.

Riguardo il risultato, esso può essere utilizzato sia per essere ritornato così per com'è oppure controllare che non sia nullo e ritornare una stringa, come nel caso dell'immagine sopra della funzione *loginQuery*.

La variabile che indica l'errore, se c'è, è un aspetto tipico della gestione delle eccezioni di Go: esiste infatti un tipo di dato chiamato *error* che viene restituito da molteplici funzioni come per esempio le conversioni, le query, le operazioni con i risultati della query, ecc... Il vantaggio è che è possibile stampare l'errore, bloccare l'esecuzione del programma con funzioni come *panic* o *log.Fatal* e tutto ciò che consegue gestire un'eccezione.

La keyword "defer" ha lo scopo di ritardare l'esecuzione di uno statement sino alla fine della funzione, quasi prima della funzione return; essa viene utilizzata per la funzione *Close()* perché impedisce ulteriori iterazioni sulle *Rows*, ovvero gli elementi ritornati dalla funzione *Query*

Se la funzione deve ritornare uno o più elementi di un certo tipo bisogna lavorare con le *Rows* che compongono il risultato dell'interrogazione effettuata ed è qui che l'utilità delle struct precedentemente definite viene in aiuto:

```

func getProfessionistiQuery(datab *sql.DB, category string) []Professionisti {
    query := fmt.Sprintf( format: "SELECT id_professionista, nome, cognome, professione, recensione, citta FR
        " WHERE professione='%s' ORDER BY recensione DESC", category)
    resp, err := datab.Query(query)
    if err != nil {
        fmt.Println(err)
    } else {
        defer resp.Close()
    }

    var lavoratore Professionisti
    var professionisti []Professionisti
    for resp.Next() {
        err2 := resp.Scan(&lavoratore.Id, &lavoratore.Nome, &lavoratore.Cognome, &lavoratore.Professione,
            &lavoratore.Recensione, &lavoratore.Citta)
        if err2 != nil {
            log.Fatal(err2)
        }

        professionisti = append(professionisti, lavoratore)
    }
    return professionisti
}

```

Funzione richiamata dall'handler della route /getprofessionisti. Restituisce una lista di professionisti iscritti a InstaFix che appartengono a una certa categoria/settore di lavoro

La variabile *resp*, che contiene il risultato della query sotto forma di Rows, viene utilizzata per due scopi: iterare gli elementi al suo interno tramite la funzione *Next* e copiare gli elementi di ogni colonna nei vari campi della variabile di tipo *Professionisti* con la funzione *Scan*; i valori di destinazione di *Scan()* come si legge dalla documentazione sono dei puntatori, per questo motivo si utilizza la notazione con *&* per indicare l'indirizzo di memoria del singolo campo di "lavoratore". Il singolo elemento di tipo *Professionisti* viene poi inserito all'interno di un array tramite la funzione *append* e a fine ciclo la lista sarà tutta inserita nell'array, che sarà l'elemento che verrà inviato tramite response HTTP al client.

4.3 Client professionista (Python)

Il client per la gestione dei professionisti è stato implementato in python in quanto questo linguaggio presenta una grande disponibilità di librerie e semplicità nell'utilizzo, oltre a implementare diversi tipi di strutture dati in modo primitivo rendendo semplice la scrittura dei programmi.

Partendo dall'aspetto più impattante del codice da un punto di vista superficiale, ovvero l'aspetto grafico, è stato gestito in questo client attraverso la libreria Tkinter e i suoi sotto moduli che rappresentano lo standard de facto per la gestione della GUI in python. Tkinter è incluso in tutte le distribuzioni Python standard ed è l'unico framework integrato nella libreria standard di Python. Questa libreria seppur leggera e semplice da usare è molto particolare nel suo funzionamento, poiché è implementata come un wrapper Python ed è gestita grazie ad un interprete Tcl. **Tcl** o **tickle** è un linguaggio di programmazione di alto livello, dinamico e interpretato proprio come python, che sfrutta diversi paradigmi di programmazione, tra i quali quello procedurale, orientato agli oggetti, imperativo e funzionale. In ogni caso non è necessario scrivere codice Tcl, perché le chiamate di Tkinter vengono tradotte in comandi Tcl grazie all'interprete incorporato. Tra i pacchetti di Tkinter utilizzati troviamo Tk e ttk:

- Ogni oggetto **Tk** incorpora una istanza dell'interprete Tcl e sfrutta la coda degli eventi di Tcl per generare ed elaborare gli eventi della GUI.
- **Ttk** nasce da una evoluzione di Tk e consente di avere degli oggetti widget più recenti che possiedono un rinnovato aspetto grafico.

Internamente, Tk e Ttk utilizzano le funzionalità del sistema operativo sottostante, ovvero Xlib su Unix/X11, Cocoa su macOS, GDI su Windows.

A meno di piccole differenze tutte le finestre hanno: strutture, funzionamenti e richieste al server molto simili. Per cui verranno riportate le parti di codici comuni a tutte le pagine e successivamente si riporteranno solo le differenze più rilevanti tra i frame.

L'entry point per l'utente che si approccia ad usare il client è costituito da una pagina di Login (login.py), o in alternativa se non si possiede ancora un account è possibile passare tramite un semplice menù alla pagina di registrazione (register.py).

La pagina di login è sicuramente una dei file più semplici del client in termini di codice ma sintetizza bene la struttura e il comportamento generale delle ulteriori pagine. Verrà presa quindi questa pagina da esempio per osservarne la struttura generale:

```
8 class LoginFrame(Frame):
9     def __init__(self, parent, controller):
10         Frame.__init__(self, parent)
11
12         buttonframe = Frame(self, highlightbackground="blue", highlightthickness=2, width=700, height=250)
13         buttonframe.pack(side="top", fill="x")
14
15         imgframe = Frame(buttonframe, highlightbackground="blue", highlightthickness=2, width=200, height=200)
16         imgframe.grid(row = 0, column = 1, pady = 10, padx = 10)
```

Definizione della classe login che ne modella la grafica e il funzionamento della finestra di login (login.py).

Come è possibile osservare dall'immagine sopra, per ogni finestra della nostra applicazione viene definita una classe in questo caso *LoginFrame* nella quale si definisce la GUI del login e anche la logica di funzionamento.

Ogni pagina dell'applicazione di fatto sarà una classe figlia di una super classe Frame importata da Tkinter. Questo perché Tkinter con la classe Frame permette di definire un contenitore per i widget della pagina.

Quindi ogni classe che modellerà la vista e il funzionamento di una pagina sarà di fatto anche un oggetto di tipo Frame e ne erediterà il comportamento.

Nella creazione di un oggetto LoginFrame viene effettuata la sovrascrizione (l'override) del metodo `__init__` della classe parent, ovvero di Frame. Di conseguenza c'è bisogno di richiamare `Frame.__init__` per assicurare che il comportamento e i metodi dell'oggetto Frame siano implementati anche nella creazione dei widget creati all'interno della classe LoginFrame.

La chiamata esplicita dell'`__init__` della classe parent è necessaria poiché python non chiama automaticamente i costruttori delle classi base, questo è utile perché

possiamo passare all'init argomenti extra come il parent.

In generale si potrebbe pensare a richiamare l'init della superclasse attraverso `super()` come vediamo nell'immagine sotto

```
class LoginFrame(Frame):  
    def __init__(self, parent, controller):  
        super(Frame, self).__init__(parent)
```

Possibile override dell'init della classe padre fattibile con classi definite in python 3.

In questo caso questo non è fattibile perché le classi di Tkinter sono classi basate su un approccio classico di python 2 nel quale non si può usare `super` ma è necessario richiamare i metodi in modo esplicito sulla classe parent. Questo approccio è superato con python 3.

Avendo così ridefinito la classe `Frame` attraverso l'override si passa all'inserimento dei widget che il frame dovrà contenere. Per ognuno di questi widget verrà specificato qual è il frame al quale verranno appesi. Questo tipicamente sarà il primo argomento per la definizione dell'oggetto widget, poi seguiranno argomenti che ne specificano lo stile.

Alla riga 31 si osserva la creazione di un ulteriore oggetto `Frame` che verrà appeso all'oggetto `self` che è l'istanza corrente di `LoginFrame`. Tutti i widget in questo caso verranno appesi dentro `frameLogin` ovvero il nuovo contenitore creato.

```
30  
31 frameLogin = Frame(self, name="frameTable", highlightbackground="red", highlightthickness=2, width=700, height=250)  
32 frameLogin.pack(expand=True, anchor=CENTER, pady=5, padx=5)  
33  
34 title = Label(frameLogin, text="Login", font=("times new roman", 20, "bold"), fg="Black").pack(side="top", anchor=CENTER,  
35  
36 f_email = Label(frameLogin, text="Email", font=("times new roman", 15, "bold"), fg="gray").pack(side="top", anchor=CENTER,  
37 self.email = Entry(frameLogin, font=("times new roman", 15), bg="lightgray")  
38 self.email.pack(side="top", anchor=CENTER, pady=5, padx=5)  
39  
40 f_password = Label(frameLogin, text="Password", font=("times new roman", 15, "bold"), fg="gray").pack(side="top", anchor=C  
41 self.password = Entry(frameLogin, font=("times new roman", 15), bg="lightgray")  
42 self.password.pack(side="top", anchor=CENTER, pady=5, padx=5)  
43
```

Widget all'interno della classe `LoginFrame` (login.py)

Le azioni per effettuare il login saranno triggerate dalla pressione di un button osservabile alla riga 61, questo andrà a lanciare una funzione di callback chiamata `login_function` (riga 44) che consentirà di lanciare una post request http verso

l'endpoint <http://localhost:8000> del server.

Per fare ciò si sfrutta un'altra importante libreria che è requests che definisce classi e metodi per effettuare richieste http. La Libreria request non rientra tra quelle standard di python ma aspira a diventarlo attraverso un'esplicita richiesta che è in fase di valutazione.

Le credenziali dell'utente (e-mail e password) vengono inserite all'interno di un dizionario che sarà trasmesso nel body della nostra POST request.

Successivamente viene indicato nell'header riga 51 quelle che sono le informazioni sul corpo della richiesta ovvero il Content-Type. Utilizziamo invece 'application/x-www-form-urlencoded' per specificare la codifica che stiamo utilizzando per il trasferimento dei dati, nello specifico questa trasporterà i dati codificati in tuple secondo il formato chiave-valore separate da '&', con un '=' tra la chiave e il valore.

I caratteri non alfanumerici sia nelle chiavi che nei valori sono codificati con il percent encoding a 8 bit.

A questo punto viene richiamato il metodo post sull'oggetto requests andando a indicare gli attributi minimi quali url,data,headers.

Nel caso in cui le credenziali siano valide il server restituirà l'Id del professionista, come si osserva in riga 55 verrà salvato in un dizionario chiamato "session".

Questo dizionario session avrà uno scope global e verrà dichiarato nel file app. Questa sarà la variabile di sessione che servirà come parametro da passare al server per effettuare numerose richieste.

In caso di credenziali errate invece, si visualizzerà un messagebox che ci avviserà del mancato login.

```

44 def login_function():
45     print("loginFunction")
46     url = 'http://localhost:8000/loginProfessionalist'
47
48     credentials = { 'password': self.password.get(),
49                     'email': self.email.get()}
50
51     headers = {'Content-Type': 'application/x-www-form-urlencoded'}
52     response = requests.post(url, data=credentials, headers=headers)
53
54     if response.text != "Credenziali errate":
55         app.session['id'] = response.text
56         app.session['logged'] = 1
57         controller.show_frame(mainpage.MainPage)
58     else:
59         messagebox.showinfo('Risultato Login', response.text )
60
61     btn_register = Button(frameLogin, text="Sign In", command=login_function, font=("times new roman",19)).pack(side="top",

```

Login_function con POST request (login.py).

Nel caso in cui non si è ancora iscritti è possibile farlo attraverso la pagina definita nel documento register.py. Questo documento non è particolarmente differente in termini di logica da quello di login appena visto. L'unica componente degna di nota è la funzione validation_form che gestisce una minuziosa sequenza di controlli per legittimare le informazioni in fase di registrazione prima di poter mandare la post request al server.

```

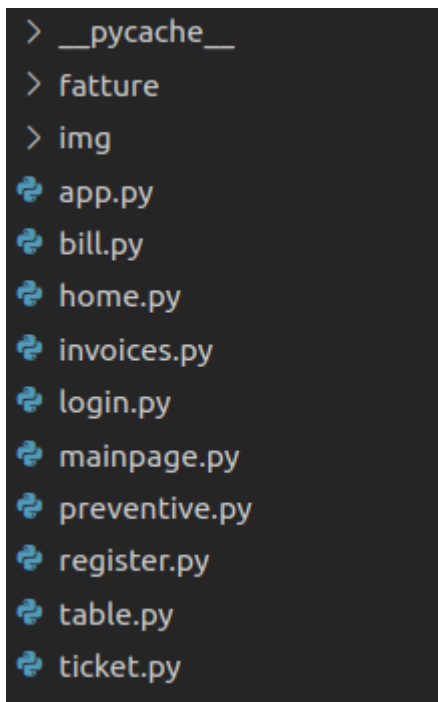
8 def validation_form(payload):
9
10     symbols = ('!', '~', '!', '@', '#', '$', '%', '^', '&', '*', '(', ')', '-', '+', '=', '{', '[', ']', '}', '|')
11     numbers = ('1', '2', '3', '4', '5', '6', '7', '8', '9', '0')
12     at = point = symbols_point = numbers_point = length_password = length_nome = length_cognome =
13
14     for i in range(len(payload['email'])) :
15         if payload['email'][i] == '@' :
16             at = at + 1
17             print("email @ = +1")
18
19         if payload['email'][-3:-2] == '.' or payload['email'][-4:-3] == '.':
20             point = point + 1
21             print("email . = +1")

```

Particolare della funzione di validazione della registrazione in register.py

Un aspetto fondamentale del client del professionista è la gestione degli oggetti Frame e la modalità con le quali queste pagine vengono rese visibili.

Questa funzionalità è inserita nel documento app.py che è il documento principale del client in quanto contiene il main. Per avere un'idea di tutti i documenti presenti è possibile osservare nell'immagine sottostante nella quale notiamo tutti i file all'interno della cartella del client.



Contenuto della cartella del client che ne contiene tutti i documenti .py e le sottocartelle.

Nel file app si trova la definizione del dizionario “session” visto in precedenza, che conterrà le informazioni di sessione.

Oltre a questo si trova la definizione della classe windows che estenderà la classe Tk di Tkinter (si nota l’override del metodo __init__ di tkinter).

La classe Tk fornisce l’accesso all’interprete Tcl e inoltre contiene comandi personalizzati per creare e manipolare i Frame e i widget della GUI.

Nella riga 22 viene definito un Frame di nome container che sarà la finestra principale che conterrà tutte le istanze delle altre classi. Questo oggetto container sarà il parametro chiamato “parent” che verrà passato per l’istanziatura di ogni classe (come visto in LoginFrame).

```
12 session = {'id': ' ',
13            'logged': 0
14            }
15
16 class windows(Tk):
17     def __init__(self, *args, **kwargs):
18         Tk.__init__(self, *args, **kwargs)
19
20         self.wm_title("INSTAFIX")
21         self.geometry("900x600+0+0")
22         container = Frame(self, height=500, width=1000, highlightbackground="red", highlightthickness=2)
23         container.pack(side="top", fill="both", expand=True)
24         container.grid_rowconfigure(0, weight=1)
25         container.grid_columnconfigure(0, weight=1)
26         self.frames = {}
```

Definizione della classe principale windows all’interno del file app.py che contiene il main.

Dal punto di vista della gestione delle classi, il codice che si trova nello screen è di fondamentale importanza in quanto nell'init del main vengono richiamate e istanziate tutte le classi del client.

Le istanze di queste classi saranno di fatto le finestre del client, che verranno tutte istanziate all'interno dell'init di windows e verranno impilate l'una sull'altra all'interno del frame container.

Ovviamente l'istanza in superficie sarà visibile. Quanto descritto è osservabile a partire dal ciclo for a riga 28, all'interno del quale iterando sulle diverse classi del client verranno di fatti istanziati i relativi oggetti con F(container,self).

In questa fase si sta implementando un paradigma Model View Controller per la gestione della parte grafica. Per fare ciò è necessario passare 3 oggetti come parametri ad ogni classe che viene istanziata, questi saranno: self, parent e controller:

- **self** rappresenta l'oggetto corrente ovvero in questo caso è l'istanza della classe windows che consente di eseguire i comandi e i widget tkinter attraverso l'interprete Tcl e di poter visualizzare i Frame e i widget descritti all'interno delle altre classi.
- **parent** rappresenta un widget che funge da genitore dell'oggetto corrente. Tutti i widget in tkinter tranne la finestra principale richiedono un genitore (a volte chiamato anche *master*). In questo caso il "parent" è costituito dall'istanza dell'oggetto frame chiamato "container" che sarà passato per l'istanziazione degli oggetti correnti, ovvero il frame "child".
- **controller** rappresenta un altro oggetto progettato per fungere da punto di interazione comune per diverse pagine di widget. È un tentativo di disaccoppiare le pagine. Vale a dire, ogni pagina non ha bisogno di conoscere le altre pagine ma se desidera interagire con un'altra pagina, ad esempio renderla visibile, può chiedere al controller di renderla visibile. Il controller sarà quindi sempre l'istanza della classe windows.

show_frame è il metodo che consente di rendere visibile una Frame o una finestra, ovvero permette attraverso il metodo tkraise() di portare in superficie un Frame

che si trova nella pila di frame. Affinché le altre classi possano richiamare il metodo `show_frame` devono chiamarlo su un'istanza della classe principale (`windows`). Tale istanza è passata a queste classi come parametro chiamato `controller`.

Alla riga 32 si può notare che il primo frame ad essere portato in superficie è il `LoginFrame`.

Infine, per questo file è presente il metodo `__main__` che si rivela essere molto semplice in quanto crea un'istanza della classe principale “`windows`”, e richiama su di essa il metodo `mainloop`.

Il metodo `mainloop` consente a Tkinter di avviare l'esecuzione dell'applicazione, come suggerisce il nome, e verrà eseguito un loop infinito fino a quando l'utente non esce dalla finestra o si ricevono eventuali eventi da parte dell'utente. Il `mainloop` riceve automaticamente gli eventi dal sistema a finestre e gli eventi dall'istanza di `windows` si propagano fino ai widget dell'applicazione.

Le applicazioni Tcl/Tk sono normalmente guidate da eventi, il che significa che dopo l'inizializzazione l'interprete esegue un ciclo infinito di eventi denominato event loop. Poiché Tkinter e quindi il `mainloop` viene eseguito su un thread singolo, i gestori di eventi devono rispondere rapidamente, altrimenti bloccheranno il thread e quindi l'elaborazione di altri eventi.

```
27
28     for F in (login.LoginFrame, register.Register, mainpage.MainPage, ticket.Ticket,
29             frame = F(container, self)
30             self.frames[F] = frame
31             frame.grid(row=0, column=0, sticky="nsew")
32     self.show_frame(login.LoginFrame)
33
34     def show_frame(self, cont):
35         frame = self.frames[cont]
36         frame.tkraise()
37
38 if __name__ == "__main__":
39     testObj = windows()
40     testObj.mainloop()
```

Main e for che consente la creazione di tutte le finestre del client (Frame) e consente di mostrarli attraverso il metodo `show_frame` (app.py).

Come visto in precedenza quando si istanzia la classe windows, che è la classe principale, vengono richiamate al suo interno tutte le altre classi che rappresentano le finestre dell'applicazione. Questo comporta che tutto il codice all'interno delle altre classi viene eseguito subito dopo l'istanziamento di windows.

Questo approccio necessario per la gestione di Tkinter porta con sé una problematica dovuta al fatto che le richieste al server per recuperare dati come informazioni sul professionista, sui ticket, sui preventivi o le fatture verrebbero durante l'istanziamento di windows. Considerando che windows viene istanziato al lancio del programma, queste richieste al server avverrebbero senza che l'utente sia ancora loggato e quindi senza le informazioni necessarie per effettuare le query.

Per evitare questo come si vede in riga 96 è stato previsto il metodo bind applicato ad uno dei frame principali della classe, in questo caso frameTable ovvero il frame che contiene la tabella.

Il metodo bind applicato su un widget consente di impostare un evento che verificatosi va ad attivare una funzione di callback (in questo caso printTicket). Nel nostro caso l'evento che triggera la callback è <expose> ovvero l'evento viene lanciato quando il widget diventa visibile.

```
frameTable.bind('<Expose>', lambda *args: MainPage.printTicket(tree, buttonframe, *args)) ]

def printTicket(tree, buttonframe, *args):
    nome = MainPage.getnome()
    l1 = Label(buttonframe, text="Benvenuto " + nome, bg="gray92", font=("times new roman", 15))
    l1.grid(row = 3, column = 0, pady = 0, padx = 0)

    jsn = MainPage.getTicketProfessionalist()

    if(len(tree.get_children()) != 0):
        for i in tree.get_children():
            tree.delete(i)

    if(jsn != None):
        total_rows = len(jsn)
        lst = ["Id", "Stato", "Categoria", "Titolo", "Descrizione"]

        for i in range(total_rows):
            if jsn[i][lst[1]] == 'creato' or jsn[i][lst[1]] == 'in attesa' or jsn[i][lst[1]] == 'in corso':
                tree.insert(parent='', index='end', iid=i, text='', values=( jsn[i][lst[0]], jsn[i][lst[2]], jsn[i][lst[3]], jsn[i][lst[4]]))
```

Bind applicato al frame e callback function necessari a non far eseguire tutto il codice all' lancio dell'applicazione.

In questo modo quando si visualizza la finestra verrà lanciato la funzione di callback, la quale conterrà altri metodi che effettueranno le request al server.

In questo caso nello screen sovrastante si osservano due metodi, `getnome()` e `getTicketProfessionalist()` che vanno a recuperare rispettivamente il nome del professionista che verrà stampato a video e i ticket afferenti al professionista. Inutile osservare che questi dati non si sarebbero potuti recuperare in fase precedenti al login.

Inoltre, nella funzione è presente un ciclo `for` che itera sugli elementi restituiti da `getTicketProfessionalist()` che saranno contenuti in un json. Queste informazioni attraverso il `tree.insert()` verranno inseriti in una tabella per essere mostrati a video.

Prima del codice appena descritto c'è un `for` che itera sugli elementi del `tree` ovvero sulle righe della tabella andandoli ad eliminare. Questo si rivela necessario poiché nel caso in cui si facciano operazioni CRUD sui ticket l'eliminazione e il reinserimento delle informazioni ci assicuriamo che la tabella (`tree`) venga aggiornata anche in istanti successivi alla prima visualizzazione.

Il metodo `getTicketProfessionalist` contiene la chiamata POST al server, essa è sicuramente la post che viene effettuata con più frequenza durante il ciclo di vita del client.

```
130 def getTicketProfessionalist():
131     print("getTicketProfessionalist")
132     url = 'http://localhost:8000/getticketsprofessionalist'
133     credentials = { 'id_professionista': app.session['id']}
134     headers = {'Content-Type': 'application/x-www-form-urlencoded'}
135     response = requests.post(url, data=credentials, headers=headers)
136
137     if response.text != None :
138         return response.json()
139     else:
140         return response.text
```

Funzione `getTicketProfessionalist` con POST request al server.

Entrambi gli spezzoni di codice appena visti si trovano nel file `mainpage.py` che contiene l'omonima classe `MainPage` che permette di visualizzare una tabella con i ticket attualmente attivi nei rispettivi stati di interesse per il professionista

“creato”, “in attesa”, e “in corso”, (i ticket finiti o votati non saranno visualizzati). Oltre a questo, è visibile un menù che permetterà la navigazione nelle diverse funzionalità del client, come:

- La visualizzazione dei nuovi ticket che hanno stato “creato” e per i quali il professionista dovrà produrre un preventivo per accettare la lavorazione. Queste funzionalità saranno rese accessibili attraverso la classe Ticket.
- La visualizzazione dei preventivi dei ticket che hanno stato “in attesa” per i quali si attende che la risposta dai clienti. Questo frame è definito attraverso la classe PreventiveAll
- La visualizzazione dei Ticket per cui si è concluso il lavoro e si trovano in stato “in corso” per i quali dovrà essere creata la fattura da mandare al cliente attraverso la classe Invoice.
- Infine, nel menu sarà presente un pulsante che richiamerà la funzione logout() responsabile di cancellare le informazioni di sessione dell’utente e il reindirizzamento al login.

```
b1 = Button(buttonframe, text="Home", command=lambda: controller.show_frame(MainPage))
b2 = Button(buttonframe, text="Nuovi Ticket", command=lambda: controller.show_frame(ticket.Ticket))
b3 = Button(buttonframe, text="Preventivi in Attesa", command=lambda: controller.show_frame(preventive.PreventiveAll))
b4 = Button(buttonframe, text="Fatturazione", command=lambda: controller.show_frame(invoices.Invoices))
b5 = Button(buttonframe, text="Logout", command=lambda: logout(controller))
```

Questo menu di navigazione sarà accessibile in ogni finestra e il funzionamento è basato sul metodo show_frame richiamato sull’oggetto controller che porta il frame richiesto in alto alla pila.

Gli ultimi aspetti interessanti del client si trovano tutti nel file bill.py che consente al professionista dopo che ha effettuato il lavoro di andare a modificare il costo del preventivo se necessario, e successivamente inviare la fattura al cliente. Questo avverrà attraverso la creazione di un PDF che conterrà le informazioni del ticket e del preventivo. Una volta creato il documento verrà fatto l’upload dello stesso in uno specifico path del server.

In particolare, questo comportamento verrà definito nella funzione insertFatturaProfessionist() sottoriportata.

```
236 test_response = requests.post(url, files = file )
```

Per la creazione del PDF si è fatto uso della libreria FPDF, rispetto ad altre librerie PDF, FPDF è semplice, leggero e versatile.

A riga 229 si osserva il nome che si dà al pdf che è costituita da una stringa molto lunga nella quale si inserisce l’id del ticket, l’id del preventivo e un numero randomico per evitare possibili anonimie dei documenti durante il testing. Per la generazione dei numeri pseudo random ci si è affidati al metodo random()

importato dall'omonima libreria che ha permesso di generare un float casuale campionato uniformemente nell'intervallo semiaperto [0.0, 1.0).

In seguito, il file viene salvato nella cartella “/fatture” del client, successivamente si recupera il file, lo si apre e lo si inserisce nel corpo della POST request diretta all'endpoint del server “/uploadfile” in modo che il server possa copiare il documento e salvarlo.

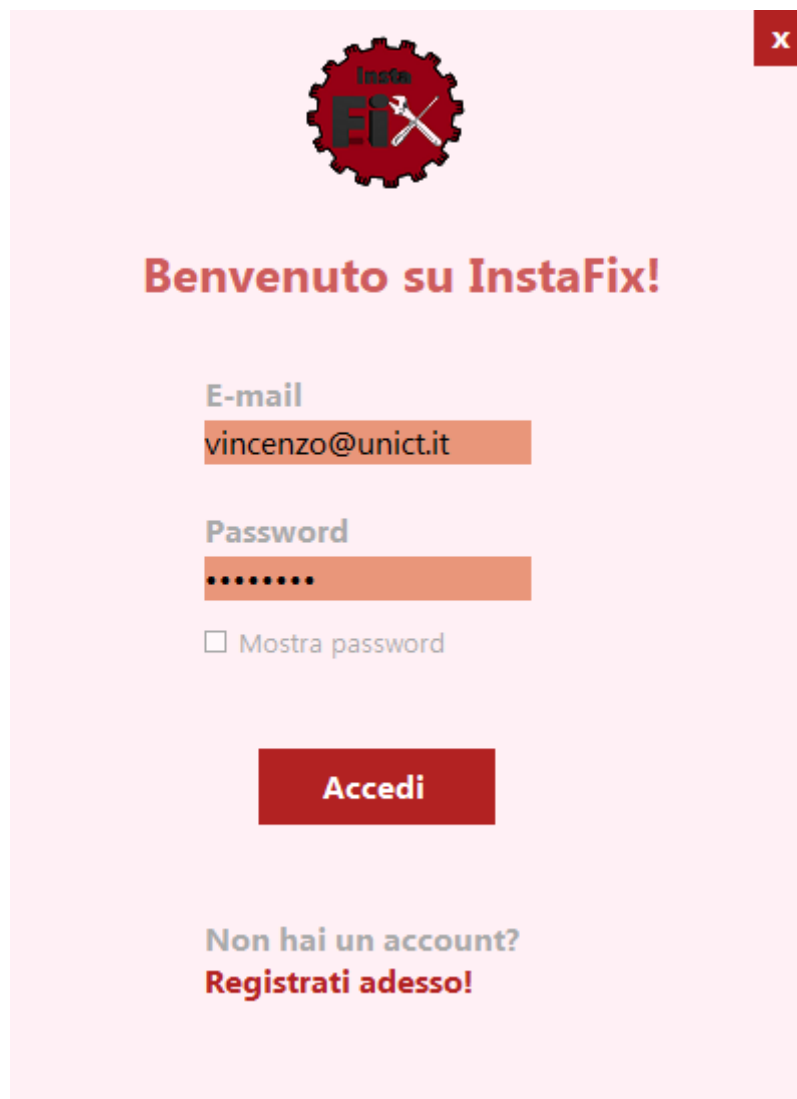
Nel codice si è fatto uso della libreria Pillow che permette all'interprete python l'elaborazione di immagini. In particolare, si è fatto uso di due moduli di questa libreria, Image e ImageTk:

- Il **modulo Image** fornisce una classe che viene utilizzata per rappresentare una PIL image. Il modulo fornisce anche una serie di funzioni per caricare immagini da file e per creare nuove immagini.
- Il **modulo ImageTk** contiene il supporto per creare e modificare oggetti Tkinter BitmapImage e PhotoImage da immagini PIL.

5 Conclusioni e risultati

Per concludere la documentazione verranno mostrati una serie di screenshot dell'applicazione evidenziando in alcuni casi messaggi di avviso o alcune eccezioni gestite. Verranno mostrati solamente i client, essendo dotati di GUI, soffermandosi maggiormente sui Windows Forms.

Per visionare il codice sorgente più nel dettaglio, si visiti la seguente [repository](#)



Benvenuto su InstaFix!

E-mail
vincenzo@unict.it

Password
.....

☐ Mostra password

Accedi

Non hai un account?
Registrati adesso!

Client utente, form di login

Registrazione

Nome

Cognome

Città

Indirizzo

E-mail

Telefono

Password

Conferma password

☐ Mostra password

Registrati

Client utente, form di registrazione

Benvenuto, Alessandro

Crea ticket


Visualizza ticket


Preventivi


Fatture


Recensioni


Client utente, homepage


Ticket


CREAZIONE TICKET




Crea ticket

Visualizza ticket

Preventivi

Fatture

Recensioni

TITOLO

Aiuole da curare


CATEGORIA

Giardinaggio

2 volte a settimana

DESCRIZIONE


Ticket OK


Il ticket è stato creato correttamente, si prega adesso di selezionare un professionista


OK


Crea


Client utente, form per creare un ticket con avviso di operazione avvenuta con successo


Ticket


SELEZIONA PROFESSIONISTA




Crea ticket

Visualizza ticket


Preventivi

Fatture

Recensioni

Nome	Cognome	Professione	Recen...	Città
Mario	Impeduglia	Giardinaggio	0,1	Catania
Luigi	Fazio	Giardinaggio	N/A	Misterbianco

Selezione

Vuoi scegliere questo specialista per il tuo ticket?

SiNo

Client utente, finestra per la selezione del professionista subito dopo aver creato il ticket

I TUOI TICKET

X

Crea ticket

Visualizza ticket


Preventivi


Fatture

Recensioni


Numero	Titolo	Categoria	Stato
1	Giardino rotto	Giardinaggio	creato
3	Clima non funzionante	Climatizzazione e r...	creato
4	pc rotto	Telecomunicazioni	creato
28	Erba da tagliare	Giardinaggio	in corso
29	Erba da curare	Giardinaggio	in attesa
30	Giardino forma	Giardinaggio	rifiutato
31	pc rotto	Telecomunicazioni	votato
32	a caso	Idraulica	creato
33	Restauro soggiorno	Edilizia	votato
34	mobile restauro	Edilizia	votato
38	Aiule da curare	Giardinaggio	creato


Client utente, form per la visualizzazione dei ticket dell'utente


Ticket





I TUOI PREVENTIVI




Crea ticket

Visualizza ticket

Preventivi

Fatture

Recensioni

Ticket di riferimento	Costo	Data & ora	Descrizione
38	50	30/08/2022 9.00	Possibilità di pulizia il ...

Client utente, lista dei preventivi inviati dai professionisti

DETAGLI DEL PREVENTIVO

Crea ticket

Visualizza ticket

Preventivi

Fatture

Recensioni

Numero preventivo

Num. ticket riferimento

Descrizione

Possibilità di pulizia il martedì e il sabato mattina

Costo

Materiali / Ricambi

Forbice e prodotti pulizia

Data e ora

Accetta

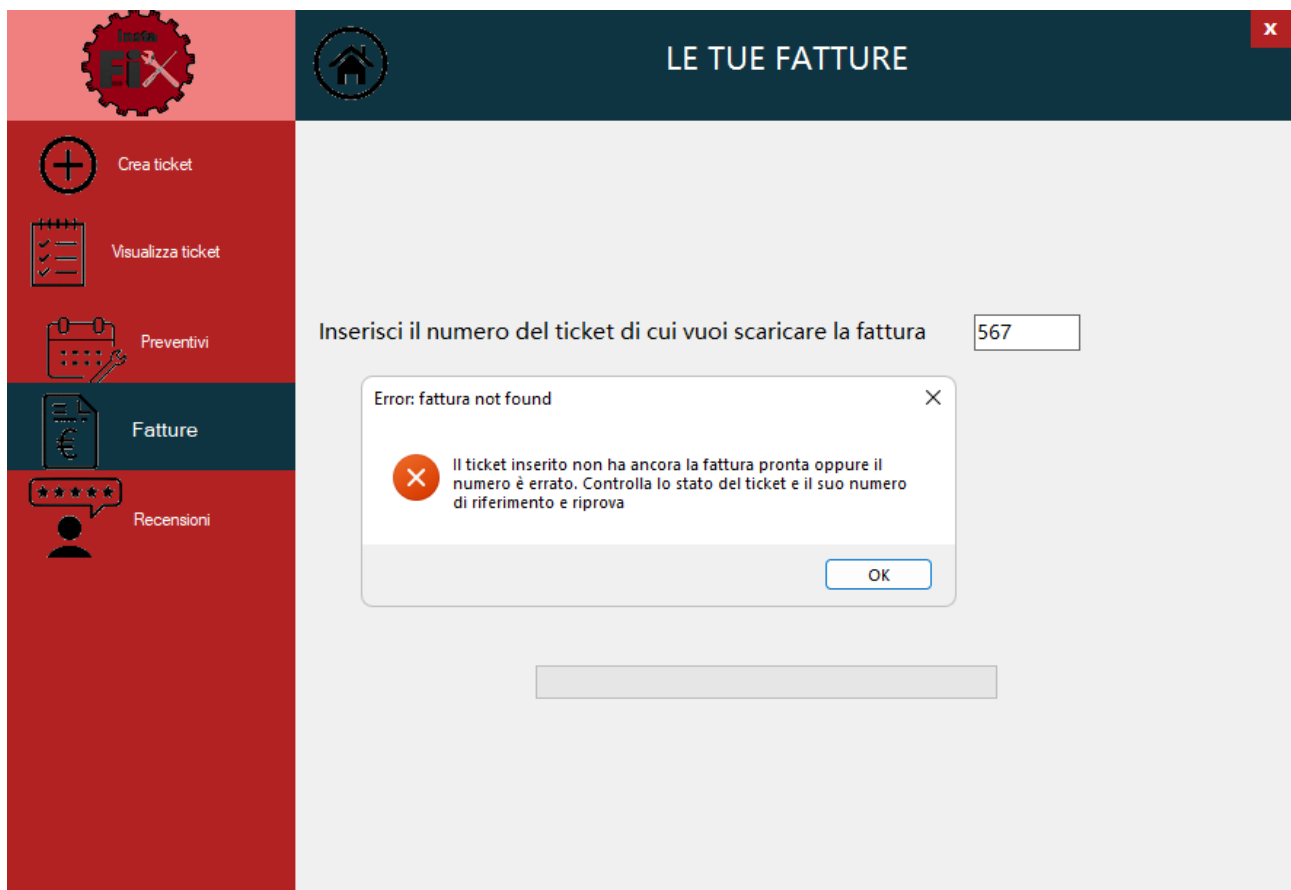
Rifiuta

Ticket in corso

Preventivo accettato: il ticket sarà concluso non appena il lavoratore, ultimato il lavoro, invierà la fattura che potrai consultare nella sezione apposita. Puoi anche lasciare una recensione a lavoro terminato

OK

Client utente: cliccando su uno dei preventivi, verrà aperto il seguente form che mostra informazioni più dettagliate sul preventivo permettendo, inoltre, di accettarlo o rifiutarlo



Client utente, finestra per scaricare le fatture con relativo popup di avviso in caso di fattura non trovata

LE TUE OPINIONI

Crea ticket

Visualizza ticket

Preventivi

Fatture

Recensioni

Seleziona il professionista da recensire

Luigi Fazio Misterbianco media recensioni= N/A

Inserisci il numero del ticket di cui vuoi valutare le prestazioni del professionista

56

Seleziona il voto

4

Vota


Error: wrong ticket or professionist

Il ticket inserito non è stato svolto dal professionista da te indicato, oppure il numero è errato. Controlla il numero del ticket nella sezione apposita e riprova

OK

Client utente, form per recensire i professionisti con popup di errore se viene inserito un ticket non effettuato dal professionista scelto

INSTAFIX



LoginRegistrati


Registrati qui

Nome	<input type="text"/>
Cognome	<input type="text"/>
Professione	<input type="text" value="Edilizia"/>
Partita IVA	<input type="text"/>
Città	<input type="text"/>
Indirizzo	<input type="text"/>
Telefono	<input type="text"/>
Email	<input type="text"/>
Password	<input type="password"/>

Sign In

Client professionista, form Registrazione.

INSTAFIX



LoginRegistrati

Login

Email


Password

Sign In

Client professionista, form Login.

INSTAFIX

HOME



Benvenuto nome

Home

Nuovi Ticket


Preventivi in Attesa

Fatturazione

Logout

Id	Stato	Categoria	Titolo	Descrizione
52	creato	Idraulica	Intervento in doccia	è necessario cambiare il box doccia
53	creato	Idraulica	rubinetto	è necessario cambiare il flessibile di un rubinetto
54	creato	Idraulica	impianto di irrigazione	si richiede un impianto d'irrigazione per un giardino
55	creato	Idraulica	lavello	montaggio lavello in una nuova cucina

Client professionista, Homepage.

INSTAFIX				
<div>  <div> Home Nuovi Ticket Preventivi in Attesa Fatturazione Logout </div> </div>				
TICKET				
Id	Stato	Categoria	Titolo	Descrizione
52	creato	Idraulica	Intervento in doccia	è necessario cambiare il box doccia
53	creato	Idraulica	rubinetto	è necessario cambiare il flessibile di un rubinetto
54	creato	Idraulica	impianto di irrigazione	si richiede un impianto d'irrigazione per giardino
55	creato	Idraulica	lavello	montaggio lavello in una nuova cucina

Client professionista, lista dei nuovi ticket aperti dai client i quali devono essere rifiutati o accettati dal professionista. L'accettazione avverrà attraverso la compilazione del preventivo.

INSTAFIX



Home

Nuovi Ticket

Preventivi in Attesa

Fatturazione

Logout

Completa il preventivo

Id	Stato	Categoria	Titolo	Descrizione
52	creato	Idraulica	Intervento in c	è necessario cambiare il box doccia

Compila il Preventivo

Descrizione Intervento	Materiali e Ricambi	Costo	Data e ora Intervento	
richiederà un intero giorno di lavoro	silicone, box doccia	120	15/9/2022 9:00	>>>

Rifiuta il Ticket

Rifiuta

Client professionista, form di compilazione del preventivo.

INSTAFIX



HomeNuovi TicketPreventivi in AttesaFatturazioneLogout

Completa il preventivo

Id	Stato	Categoria	Titolo	Descrizione
54	creato	Idraulica	impianto di irr	si richiede un impianto d'irrigazione per giardino

Risultato Inserimento

 **Inserito correttamente**

OK

Descrizione Intervento	Materiali e Ricambi	Costo	Data e ora Intervento	
richiederà due giorni di lavoro	tubo e spruzzi	300	14/09/2022	>>>

Rifiuta il Ticket

Rifiuta

Client professionista, preventivo compilato e inserito correttamente.

INSTAFIX



Home

Nuovi Ticket

Preventivi in Attesa

Fatturazione

Logout

Completa il preventivo

Id	Stato	Categoria	Titolo	Descrizione
55	creato	Idraulica	lavello	montaggio lavello in una nuova cucina

Risultato Inserimento

 **Ticket rifiutato**

OK

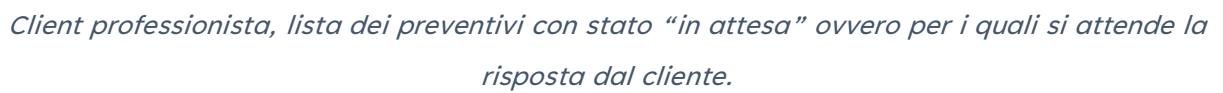
Descrizione Intervento	Materiali e Ricambi	Costo	Data e ora Intervento

>>>

Rifiuta il Ticket

Rifiuta

Client professionista, ticket rifiutato dal professionista.



INSTAFIX



[Home](#)[Nuovi Ticket](#)[Preventivi in Attesa](#)[Fatturazione](#)[Logout](#)

Fatture in corso

selezionare la fattura da mandare al cliente

Id	Stato	Categoria	Titolo	Descrizione
52	in corso	Idraulica	Intervento in doccia	è necessario cambiare il box doccia

Client professionista, lista fatture dei ticket con stato “in corso” per i quali il professionista deve compilar la fattura.

INSTAFIX



Home
Nuovi Ticket
Preventivi in Attesa
Fatturazione
Logout

Fatturazione

Ticket

Id	Stato	Categoria	Titolo	Descrizione
52	in corso	Idraulica	Intervento in doccia	è necessario cambiare il box doccia

Preventivo

Descrizione	Materiali/Ricambi	Costo	Data e Ora
richiederà un intero giorno di lavoro	silicone, box doccia	120	15/9/2022 9:00

Crea e Invia la Fattura

Crea Fattura


Aggiorna Costo Intervento

120

>>>

Client professionista, form fatturazione.

INSTAFIX



Home
Nuovi Ticket
Preventivi in Attesa
Fatturazione
Logout

Fatturazione

Ticket

Id	Stato	Categoria	Titolo	Descrizione
52	in corso	Idraulica	Intervento in doccia	è necessario cambiare il box doccia

Risultato Inserimento

inserito correttamente

OK

Descrizione	Materiali/Ricambi	Costo	Data e Ora
richiederà un intero giorno di la		120	15/9/2022 9:00

Crea e Invia la Fattura

Crea Fattura


Aggiorna Costo Intervento

125

>>>

Client professionista, aggiornamento del costo dell'intervento poiché il preventivo era stato ottimistico.

INSTAFIX



[Home](#)[Nuovi Ticket](#)[Preventivi in Attesa](#)[Fatturazione](#)[Logout](#)

Fatturazione

Ticket				
Id	Stato	Categoria	Titolo	Descrizione
52	in corso	Idraulica	Intervento in doccia	è necessario cambiare il box doccia

Descrizione

richiederà un intero giorno di la


Costo

120

Data e Ora

15/9/2022 9:00

Risultato Inserimento

**Fattura creata e inviata correttamente**

[OK](#)

Crea e Invia la Fattura

[Crea Fattura](#)

Aggiorna Costo Intervento

[>>>](#)

Client professionista, fattura pdf correttamente compilata e inviata al cliente.