Struttura del documento

1. Sommario
2. Introduzione
3. Indice
4. Capitoli
5. Conclusioni
6. Bibliografia
7. Appendici

Sommario

Implementazione di un pannello di controllo
per la gestione di un'azienda

In English:	In italiano:
In my project I wanted to include three keywords: Management, Control Panel and Company. The keyword Company invokes the subject of my project, the entity to which I have related with the goal of improving it, to promote and facilitate control, to help companies, from small start-ups to multinationals, especially in the B2C environment. The second keyword is Management, because it offers monitoring of company activities, such as customer management, profits, etc., which I will be going to describe in more detail later, with the possibility of "real-time" operations on the data. The third keyword is Control Panel, better known as Dashboard, is the main subject of the website I have developed, especially thanks to the very "user-friendly" user interface. I have worked on this project using the skills I learned at school for two-thirds of the project, for the database part and the communication between front-end and back-end, while the remaining one-third I learnt in the PCTO.	In questo mio progetto ho voluto inserire tre parole chiave: Gestione, Pannello di controllo e Azienda. La parola chiave **Azienda** richiama il soggetto del mio progetto, l'ente a cui ho fatto riferimento con l'obiettivo di migliorarlo, così da promuovere e facilitare il controllo, di agevolare le aziende, dalla piccola start-up alla multinazionale, soprattutto nell'ambito **B2C**. La seconda parola chiave è **Gestione**, perché offre il monitoraggio delle attività aziendali, come la gestione clienti, profitti, ecc. che in seguito andrò a descrivere più nel dettaglio, con possibili **operazioni "real-time"** sui dati. La terza parola chiave è **Pannello di controllo**, in inglese più conosciuta come Dashboard, è il tema principale del sito web che ho implementato, soprattutto grazie all'interfaccia utente molto "**User-friendly"**[1]. Questo lavoro lo ho sviluppato con le competenze apprese a scuola per del progetto, per la parte database e la comunicazione tra front-end e back-end, mentre il restante appreso in PCTO.

[1] "User-friendly | Definition of User-friendly by Merriam-Webster." https://www.merriam-webster.com/dictionary/user-friendly

Introduzione

Ho realizzato questo tema per interesse personale, iniziando a informarmi sulle opportunità e idee su che cosa fare dopo la maturità, perché ho deciso di non proseguire gli studi. In queste ricerche mi sono imbattuto in dei video in cui parlavano di business, soprattutto digitale, evidenziando le opportunità che il progresso digitale può portare.

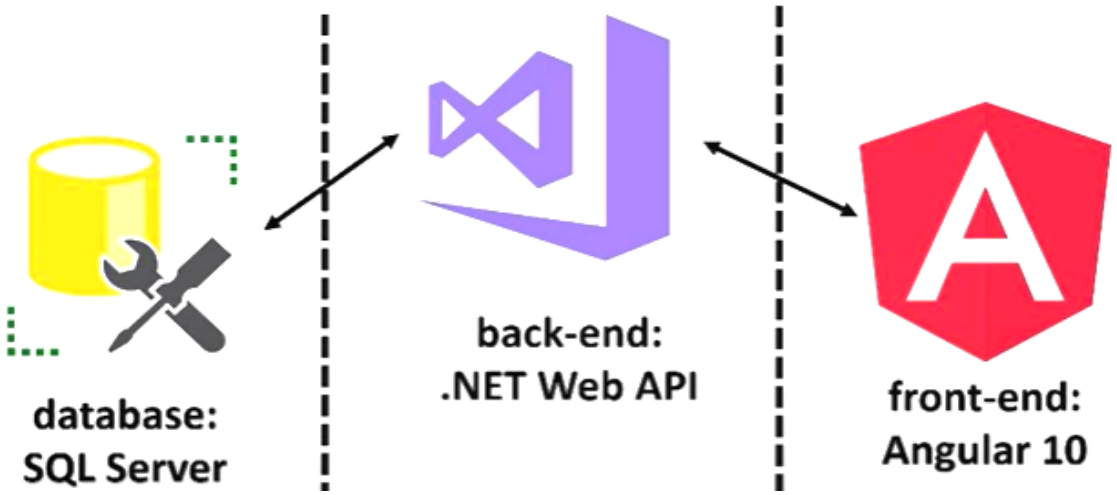
Ovviamente ne deriva il fattore monetario, come per esempio le spese in caso di apertura della partita IVA, potenziali spese aggiuntive impreviste, e molte altre considerazioni, che mi hanno portato a cercare di produrre una semplificazione, anche se già presente sul mercato una versione simile, ho avuto la possibilità di vederlo in prima persona grazie al prof. Fantini che ha organizzato un incontro in cui il CEO di Rilheva, un'azienda italiana che cerca di prevedere e risolvere dei problemi aziendali che prima erano imprevedibili. Dopo questa esperienza ho deciso cosa avrei realizzato per la mia maturità ed ho iniziato ad approfondire sempre di più l'argomento, stando attento a mantenere la coerenza con le conoscenze apprese in questi anni e senza andare fuori contesto e con la possibilità di implementazione futura.

Indice

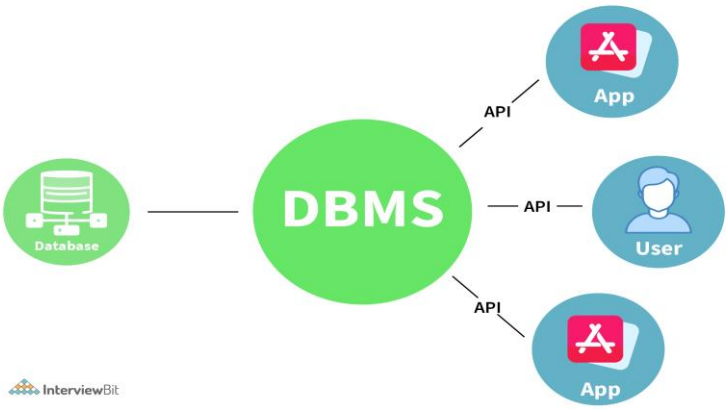
Capitoli

Capitolo 1 - Definizione componenti e struttura progetto

Questa immagine descrive perfettamente i componenti e la struttura dell'applicazione che ho costruito, con principalmente tre componenti

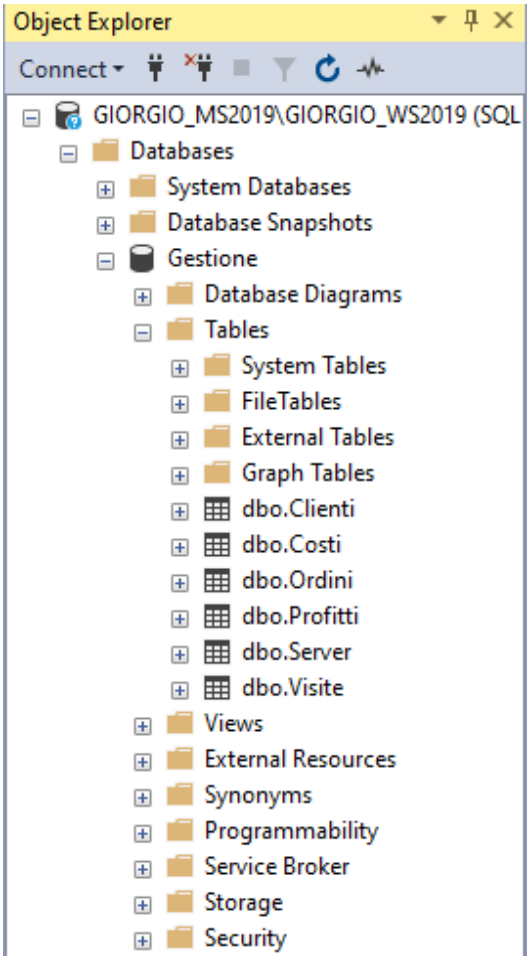


1. Database[2] (SQL Server)
   Per la base di dati ho scelto l'utilizzo di SQL Server, un Database Management System[3] prodotto da Microsoft di tipo relazionale: è un software progettato per consentire la creazione, la manipolazione e l'interrogazione di database, cioè il gestore o motore del database. È il componente da cui ho iniziato, ed ho progettato la memorizzazione di dati su tabelle, assicurandomi il corretto tipo di dato per ogni campo che ho inserito nelle tabelle.
   Il linguaggio di questo componente è SQL.



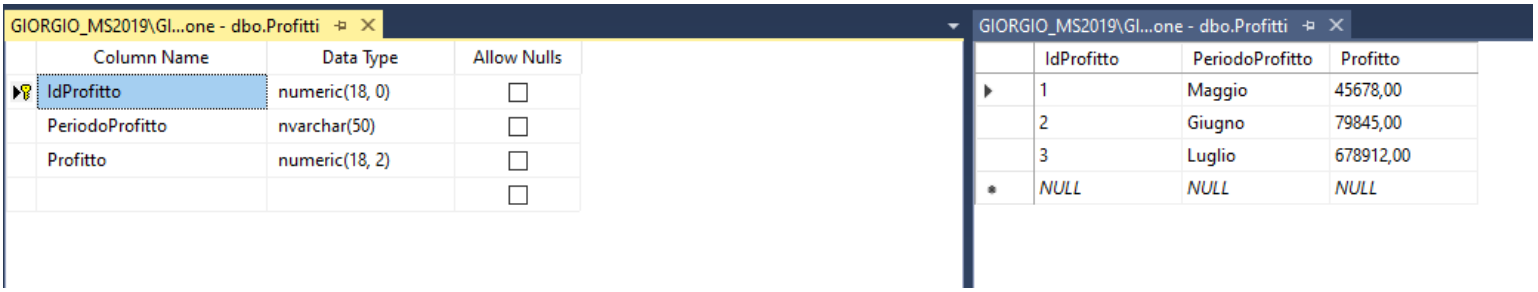
[2] "Database - Wikipedia." https://en.wikipedia.org/wiki/Database.
[3] "What is DBMS (Database Management System)?"https://www.guru99.com/what-is-dbms.html.

Per poter utilizzare il database ho usufruito di SQL Server Management Studio (SSMS), che è un ambiente integrato per la gestione di qualsiasi infrastruttura SQL, da SQL Server al database SQL di Azure.
SSMS offre gli strumenti per configurare, monitorare e amministrare le istanze di SQL Server e i database. Usare SSMS per distribuire, monitorare e aggiornare i componenti del livello dati usati dalle applicazioni, nonché per creare query e script.
È possibile usare SSMS per eseguire query, progettare e gestire database e datawarehouse in qualsiasi posizione, nel computer locale o nel cloud.
In questo ambiente possiamo anche visualizzare tutte i database e tabelle a lui associate, anche i dati presenti in esse e la loro struttura (che affronteremo in seguito).



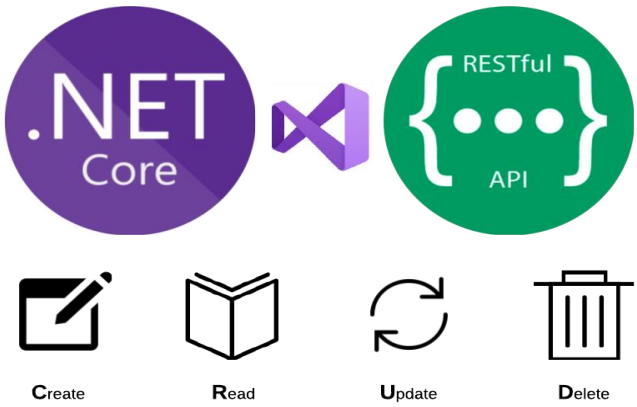
A sinistra: il database Gestione con le tabelle create per l'app Dashboard, dopo aver eseguito l'accesso con le credenziali corrette.

In basso: struttura e dati della tabella Profitti visibili in SSMS

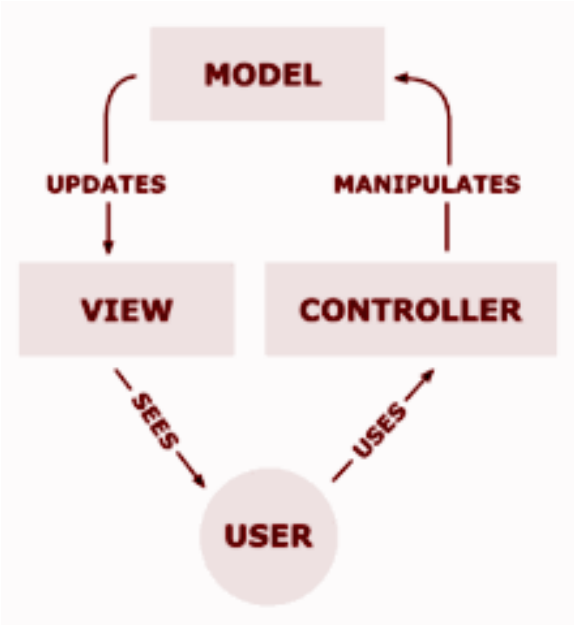


2. .NET Web API

Questo è il componente comunicante tra le due parti, ed è sviluppato in ASP.NET Core[4], un framework Web sviluppato da Microsoft, che noi studenti abbiamo appreso in laboratorio attraverso l'IDE[5] Visual Studio 2019. Esso è nato per dare la possibilità agli sviluppatori di creare app web, attraverso lo schema MVC, un framework di presentazione, ottimizzato per l'uso con ASP.NET Core, che divide l'applicazione in tre parti: Model, View, Controller.
Questo schema è basato su un sistema che separa le competenze: così facendo si agevola il modello aziendale, cioè la separazione delle problematiche, così da poter sviluppare più parti di codice da più team di sviluppo. Grazie a MVC per lo sviluppo di web app, come questo progetto, i Controller svolgono la funzione di API Web.
Il linguaggio utilizzato è C#. I tipi di dati nei due altri componenti devono corrispondere, o comunque essere compatibili tra di loro.



A sinistra : introduzione al concetto CRUD, approfondito in seguito, caratteristica delle API in .NET Core



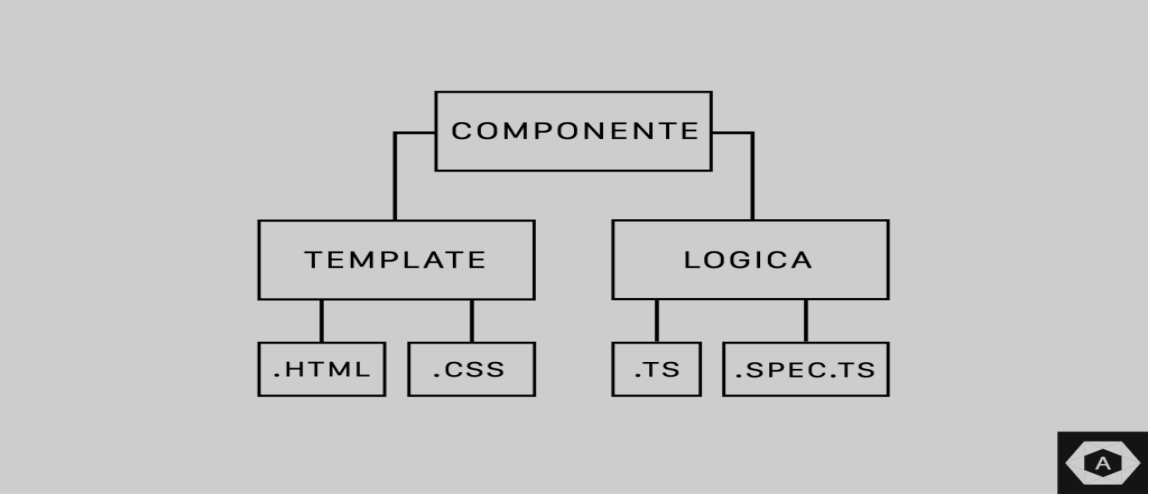
A destra:
struttura dello schema MVC, con la suddivisione in parti

[4] "What is ASP.NET Core? | .NET." https://dotnet.microsoft.com/learn/aspnet/what-is-aspnet-core.
[5] "Integrated development environment - Wikipedia." https://it.wikipedia.org/wiki/Integrated_development_environment.

3. Angular 10

Angular, evoluzione di Angular, sviluppato da Google, è un framework [6] Javascript open source, progettato per fornire uno strumento di sviluppo applicazioni web adatte a qualunque piattaforma, smartphones, tablet e computer.
Ho scoperto questo framework grazie all'alternanza scuola-lavoro che ho eseguito nel mese di febbraio, dove ho appreso le basi che mi hanno aiutato ad ottenere un buon risultato rispettando le scadenze per questo elaborato.
Per capire meglio questo componente è necessario approfondire dei concetti:
il linguaggio utilizzato è TypeScript, che non abbiamo approfondito a scuola, ed esso viene utilizzato nei componenti, che sono delle classi in Javascript che forniscono al componente la vista per visualizzare i metadati della classe.
Ogni componente ha quattro file per il funzionamento, divisibili in due categorie:

- Categoria Template:
  - file CSS → questo file serve per l'aspetto grafico della pagina
  - file HTML → dove è presente la struttura della pagina
- Categoria Logica
  - file .ts → per lo sviluppo delle funzioni del componente
  - file .spec.ts → esegue i test di funzionamento

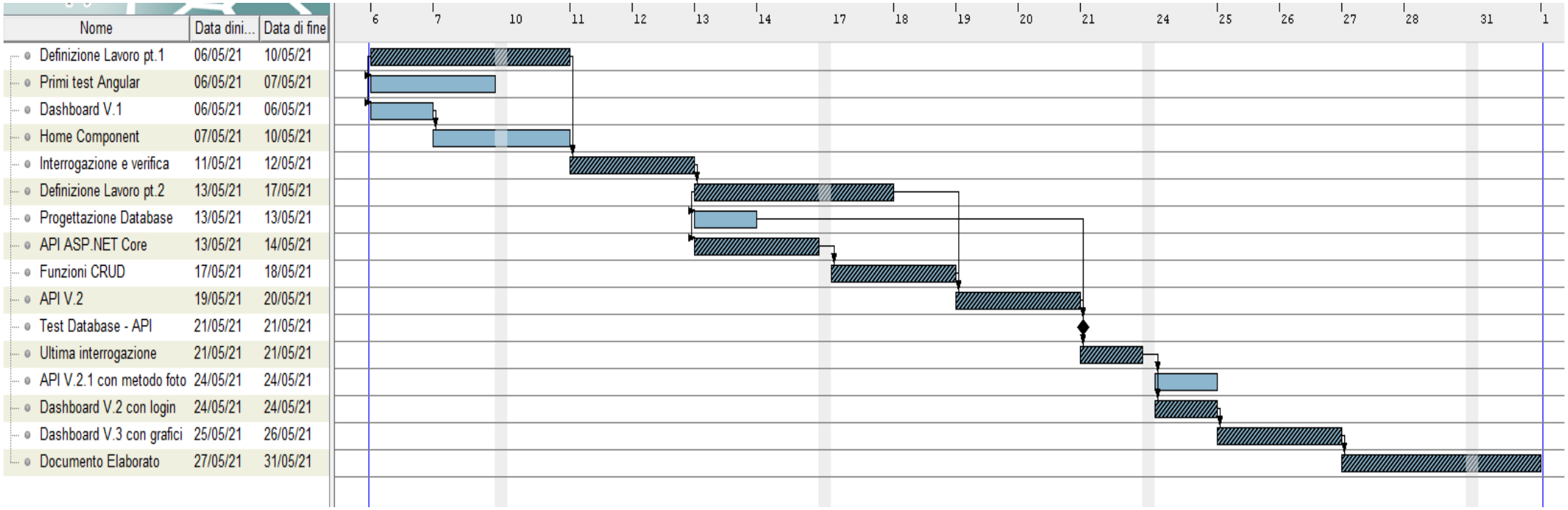


Mentre la seconda base è quella dei moduli, cioè delle classi che permettono di organizzare l'applicazione, creando strutture che contengono entità che hanno relazioni fra loro.
In questo modo possiamo raggruppare all'interno di un singolo contenitore di componenti.
Molto importante a mio avviso è il file ***routing.module.ts*** che serve per l'organizzazione delle pagine del sito, in cui a seconda dell' URL decidiamo quale componente caricare.
Inoltre ogni componente ha il metodo `ngOnInit(),` che è paragonabile al Page_Load di ASP.NET, cioè il primo metodo che viene eseguito quando il componente viene richiamato.

[6] "Framework - Wikipedia." https://it.wikipedia.org/wiki/Framework

Capitolo 2 - Progettazione

In questa sezione ho aggiunto il diagramma di Gantt, approfondito nelle lezioni di GPOI, che traccia la durata e la dipendenza delle attività che ho svolto. Per approfondire nel dettaglio ho aggiunto anche due attività scolastiche.



Capitolo 3 - Sviluppo Database

Come menzionato prima, il database che si utilizza deve avere una base di dati solida per essere utilizzata anche nel tempo, ed in questo progetto io ho pensato a sei tabelle che possono includere le principali caratteristiche che una gestione aziendale deve avere.

Clienti

Nome Campo	Tipo di dato
IdCliente (PK)	numeric(18/0)
Nome	nvarchar(50)
Cognome	nvarchar(50)

Costi

Nome Campo	Tipo di dato
IdSpesa(PK)	numeric(18/0)
Tipo	nvarchar(50)
Importo	numeric(18/2)

Profitti

Nome Campo	Tipo di dato
IdProfitto (PK)	numeric(18/0)
PeriodoProfitto	nvarchar(50)
Profitto	numeric(18/2)

Server

Nome Campo	Tipo di dato
IdServer (PK)	numeric(18/0)
StatusServer	bit

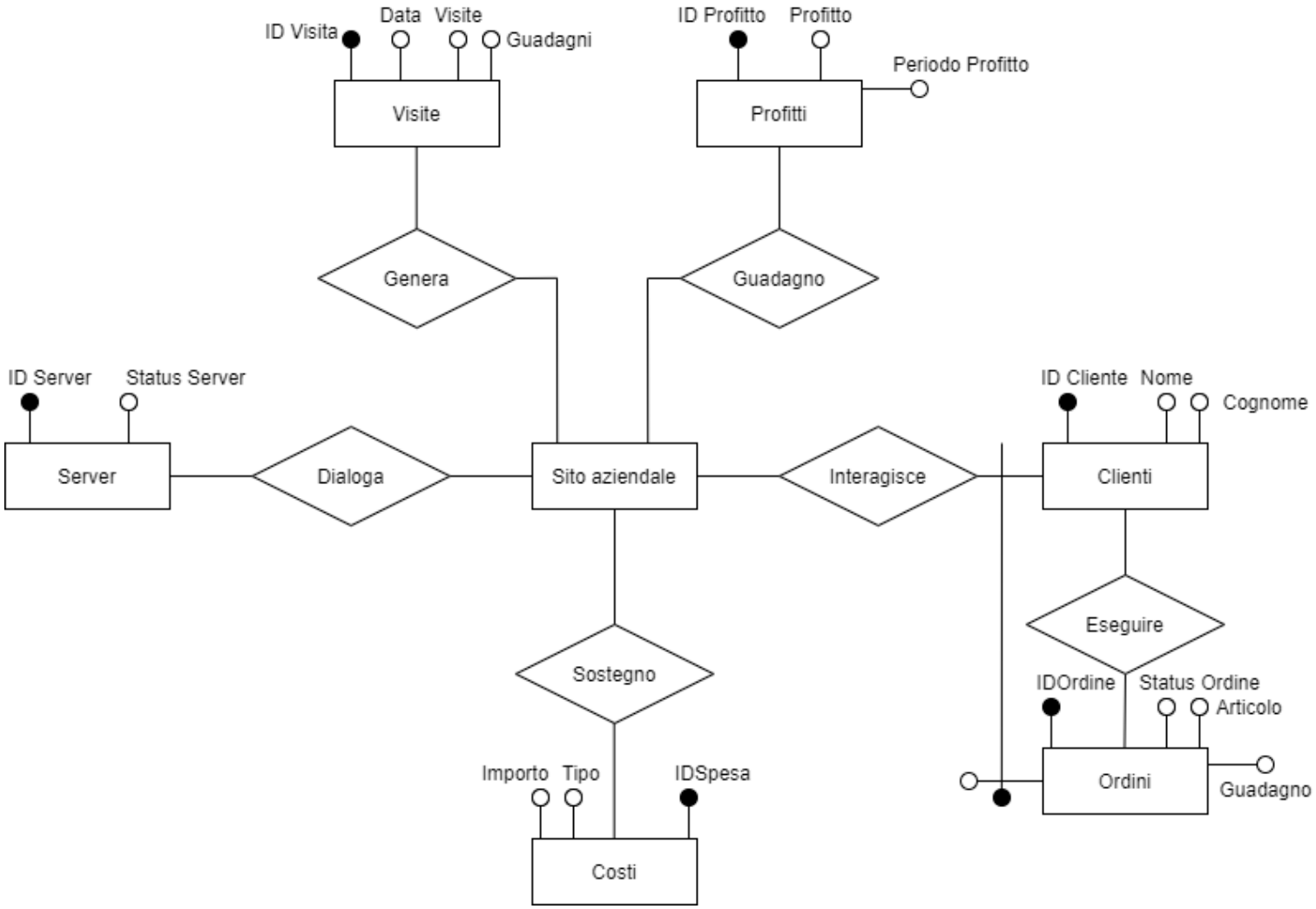
Visite

Nome Campo	Tipo di dato
IdVisite (PK)	numeric(18/0)
Data	date
Nvisite	numeric(18/0)
GuadagniPagina	numeric(18/2)

Ordini

Nome Campo	Tipo di dato	Foreign Key
IdOrdine (PK)	numeric(18/0)	
NumCliente (FK)	nvarchar(50)	FK al campo IdCliente della tabella Clienti
StatusOrdine	bit	
Articolo	nvarchar(50)	
Guadagno	numeric(18/2)	

Come abbiamo potuto apprendere in classe ogni ad database corrisponde una sua progettazione, e per quanto riguarda la progettazione concettuale ho fatto questo schema E-R (entity-relationship) che rappresenta le associazioni (forma di rombi) con le entità (quadrati con lo stesso nome delle tabelle nella pagina precedente).
I puntini neri rappresentano le chiavi primarie, quelli vuoti gli attributi e l'intreccio di un attributo con un pallino nero rappresenta la Foreign key (chiave esterna) che fa riferimento ad un'altra tabella. Nello schema questo esempio è raffigurato nell'entità Ordini.



Capitolo 4 - API ASP.NET Core

Prima di introdurre gli elementi dell'API bisogna fare una precisazione sulla modifica di impostazioni nei file di esecuzione di essa per comprendere tale modifica che eccezioni gestisce.
Per l'esecuzione del programma vengono utilizzati due file: Program.cs e Startup.cs .
Il primo crea un'istanza ed esegue il secondo, che serve come "configurazione iniziale".
In Startup.cs bisogna modificare i Cors : Cors Origins Resources Sharing, che bloccano le richieste da diversi domini. Così facendo tutte le richieste che riceve l'API vengono considerate e nessuna scartata.

```
public class Program
{
    0 riferimenti
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    1 riferimento
    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

A sinistra: codice del file Program.cs

In basso: codice del file Startup.cs

```
2 riferimenti
public class Startup
{
    0 riferimenti
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    1 riferimento
    public IConfiguration Configuration { get; }
    0 riferimenti
    public void ConfigureServices(IServiceCollection services)
    {

        // Abilitazione Cors, perché normalmente la policy di default impedisce le richieste da diversi origini, domini, metodi
        services.AddCors(c =>
        {
            c.AddPolicy("AllowOrigin", opzioni => opzioni.AllowAnyOrigin().AllowAnyMethod().AllowAnyHeader());
        });

        EnableCorsAttribute cors = new EnableCorsAttribute();
        services.AddControllersWithViews().AddNewtonsoftJson(opzioni => opzioni.SerializerSettings.ReferenceLoopHandling = Newtonsoft.Json.ReferenceLoopHandling.Ignore)
            .AddNewtonsoftJson(opzioni => opzioni.SerializerSettings.ContractResolver = new DefaultContractResolver());

        services.AddControllers();
    }
```

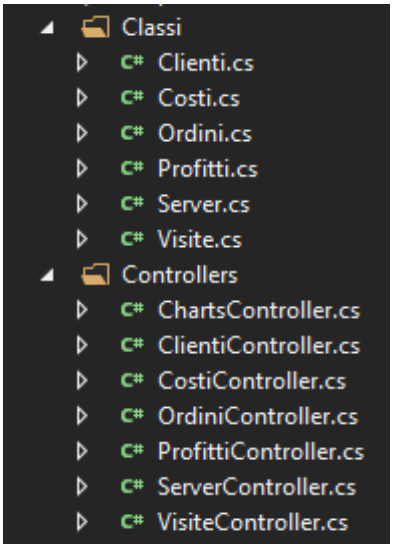
La comunicazione

Per ottenere la comunicazione ogni tabella del database è rappresentata da una classe con la stessa denominazione in un file .cs con i nomi identici dei capi della tabella con il tipo di dato più compatibile possibile. Qui sotto un esempio della tabella Clienti:

Clienti

Nome Campo	Tipo di dato
IdCliente (PK)	numeric(18/0)
Nome	nvarchar(50)
Cognome	nvarchar(50)

```
2 riferimenti
public class Clienti
{
    2 riferimenti
    public int IdCliente { get; set; }
    2 riferimenti
    public string Nome { get; set; }
    2 riferimenti
    public string Cognome { get; set; }
}
```




```
[Route("api/[controller]")]
[ApiController]
1 riferimento
public class ClientiController : ControllerBase
{
    private readonly IConfiguration configuration;
    private readonly IWebHostEnvironment envirionment;
    0 riferimenti
    public ClientiController(IConfiguration config, IWebHostEnvironment env)
    {
        configuration = config;
        envirionment = env;
    }
```

Sopra: ClientiController, è il nome del file controller che riferisce alla classe clienti, esso viene richiamato con il link “api/clienti”

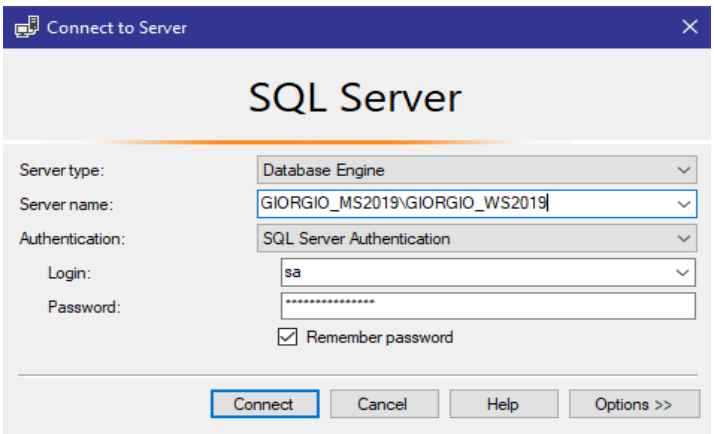
Ognuna di queste classi ha associato un controller, come descritto in precedenza nello schema MVC, in cui sono presenti i metodi del protocollo applicativo HTTP, nello specifico i metodi che compongono CRUD[7], cioè le operazioni basiche per interagire con le tabelle del nostro database, per questo servono i metodi GET (richiesta di risorse), POST (inserimento di dati), PUT (aggiornamento dei dati) e DELETE (cancellazione di dati).
Questi controller vengono richiamati con l’url “api/controller“, dove controller è il nome del file prima di Controller.cs.
Le variabili read-only configuration ed environment servono al funzionamento del controller, sono applicate al costruttore di default passati come parametri.
Nella pagina precedente si può notare nel file Startup.cs la presenza di Configuaration

[7] "Tavola CRUD - Wikipedia." https://it.wikipedia.org/wiki/Tavola_CRUD.

```
[HttpGet]
0 riferimenti
public JsonResult GetJson()
{
    //Ultimo accesso bisogna convertirlo per farlo funzionare
    string query = @"SELECT IdCliente, Nome, Cognome FROM dbo.Clienti";
    DataTable tabella = new DataTable();
    // Configurazione con elementi ADO come visto in laboratorio
    string strcn = configuration.GetConnectionString("MyConn");
    SqlDataReader dr;
    using (SqlConnection cn = new SqlConnection(strcn))
    {
        cn.Open();
        using (SqlCommand cmd = new SqlCommand(query, cn))
        {
            dr = cmd.ExecuteReader();
            tabella.Load(dr);
            cn.Close();
        }
    }
    //vedo e ritorno il risultato nel data table come JSON
    return new JsonResult(tabella);
}
```

A sinistra:
metodo GET contenuto nel controller Clienti

Dopo aver dichiarato quale metodo sia, indicato con [HttpGet], si usa la classe JsonResult per semplificare la comunicazione tra le parti, utilizzando JSON (JavaScript Object Notation), che è un formato adatto per lo scambio di dati fra applicazioni client/server.
La stringa query è l'interrogazione che l'API farà al database dopo una specifica azione compiuta dall'utente.
DataTable, strcn, dr e cn sono tutti elementi ADO.NET, un set di classi Microsoft che instaurano una connessione logica tra database e API.
Ho utilizzato using perché in questo modo sono sicuro che l'esecuzione del programma va a buon fine, senza avere delle condizioni specifiche.
Prima di tutto bisogna aprire una connessione, una sorgente di dati per far funzionare il tutto.
Una volta aperta bisogna dichiarare un comando, che utilizza la connessione con una query per interrogare il database, che in questo caso vorrebbe prendere i dati della tabella per farli visualizzare all'utente.
I dati della tabella vengono letti da dr, il Data Reader, che è un oggetto connessione readonly, che recupera i record restituiti dalla query.
Questi record vengono inseriti in tabella, oggetto non connesso, che viene utilizzato per rappresentare le tabelle in un set di dati, ed infatti è l'elemento che viene restituito dalla funzione sotto forma di JSON (ultima riga).
Tutto questo è possibile alla variabile cn, la connessione ADO che si connette al database, questa stringa sarà presente nel file di progetto denominato appsettings.json.
Questa stringa di connessione è visibile anche nel programma Microsoft SQL Server Management Studio, come riportato qui sotto.



A sinistra:
schermata iniziale quando apriamo SSMS, con le credenziali da inserire. È importante che queste credenziali siano uguali in entrambi le parti, altrimenti nessun dato verrà visualizzato dall'utente.

```
"ConnectionStrings": {
  "MyConn": "Data Source=sqlDataSource;Server=GIORGIO_MS2019\\GIORGIO_WS2019;Database=Gestione;User Id=sa;Password=trDPKfd3.k;"
  //"MyConn": "Data Source=sqlDataSource;Server=C212-07;Database=Esame;User Id=5G07;Password=studente1;"
},
```

In basso: stringa presente nel file appsettings.json

Questa stringa di connessione è uguale per tutti i metodi http, mentre cambiano la query, utilizzando gli attributi della classe, ed il risultato che il metodo restituisce sotto forma di JSON.

Ecco un esempio di POST nel controller Costi:

```
[HttpPost]
0 riferimenti
public JsonResult Post(Costi cos)
{
    string query = @"INSERT INTO dbo.Costi (IdSpesa,Tipo,Importo) VALUES (" + cos.IdSpesa + @",'" + cos.Tipo + @"', " + cos.Importo + @")";
    DataTable tabella = new DataTable();
    // Configurazione con elementi ADO come visto in laboratorio
    string strcn = configuration.GetConnectionString("MyConn");
    SqlDataReader dr;
    using (SqlConnection cn = new SqlConnection(strcn))
    {
        cn.Open();
        using (SqlCommand cmd = new SqlCommand(query, cn))
        {
            dr = cmd.ExecuteReader();
            tabella.Load(dr);
            cn.Close();
        }
    }
    //vedo e ritorno il risultato nel data table come JSON, è il messaggio che vedrà l'utente sotto forma di alert
    return new JsonResult("inserimento dati (POST) andato a buon fine");
}
```

Lo stesso ragionamento può essere applicato ai metodi PUT e DELETE, visibile qui sotto per il passaggio parametri direttamente dall'URL, per ottimizzare il processo di cancellazione (sempre riferito alla tabella costi).

```
[HttpDelete("{id}")]    //senza id non lo prenderebbe dall'url
0 riferimenti
public JsonResult Delete(int id) //riceve direttamente l'ID utente per ottimizzazione
{
    string query = @"DELETE FROM dbo.Costi WHERE IdSpesa='" + id + @"'";
    DataTable tabella = new DataTable();
    // Configurazione con elementi ADO come visto in laboratorio
    string strcn = configuration.GetConnectionString("MyConn");
    SqlDataReader dr;
    using (SqlConnection cn = new SqlConnection(strcn))
    {
        cn.Open();
        using (SqlCommand cmd = new SqlCommand(query, cn))
        {
            dr = cmd.ExecuteReader();
            tabella.Load(dr);
            cn.Close();
        }
    }
    //vedo e ritorno il risultato nel data table come JSON
    return new JsonResult("cancellazione dati (DELETE) andato a buon fine");
}
```

Capitolo 5 - Interfaccia: Angular

Per l'interfaccia come IDE ho utilizzato Visual Studio Code, ottimo per lo sviluppo di applicazioni in ambito web.
Per iniziare a sviluppare un'applicazione Angular bisogna assicurarsi di aver installato Node.js,un Runtime system open source multipiattaforma orientato agli eventi per l'esecuzione di codice JavaScript, che crea la cartella node_modules, che contiene tutte le librerie che il framework utilizza, scaricandole con

```
>npm install nome_pacchetto
```

Solo in seguito installare la CLI di angular con il comando.

```
cd C:\Users\m_inf_5g\Desktop> npm install -g @angular/cli
```

Il comando npm è il più grande ecosistema di librerie open-source.

Dopodiché per creare la base del progetto basta digitare

```
>ng new Nome Progetto
```

In questo punto bisogna scegliere il tipo di linguaggio di aspetto da usare (CSS,SCSS)

Per generare un componente, cioè una parte del sito che si sta costruendo bisogna digitare

```
> ng generate component Nome Componente
```

Mentre per il modulo (come introdotto precedentemente) basta modificare il comando in

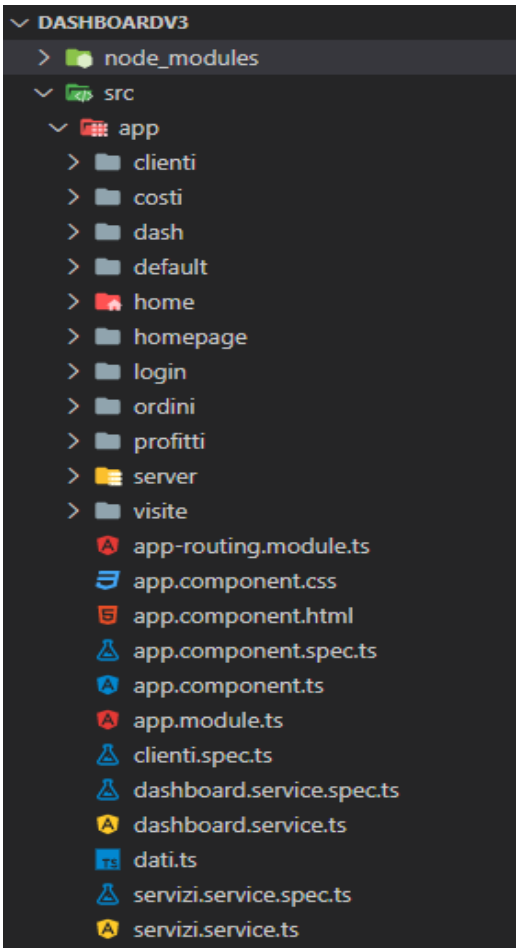
```
> ng generate module Nome Componente
```

Dopo aver creato il progetto basta compilare nella cartella del nuovo progetto

```
C:\Users\m_inf_5g\Desktop\Progetto>ng serve
```

eseguendo ng serve si compila i file predefiniti, chiamati index.html, styles.css e main.ts, per tenere in esecuzione il programma. Ogni volta che si modifica uno dei file presenti nel progetto, in automatico la console compilerà di nuovo questi file con le modifiche aggiunte, ed in caso di errore ne individua la causa e lo descrive nella pagina, segnalandolo come un errore di compilazione.
Questa era un'introduzione per l'installazione del progetto, nel prossimo capitolo descriverò il mio progetto personale, con la suddivisione delle parti e parti fondamentali per il suo corretto funzionamento.

Capitolo 6 - Tabelle



A sinistra:
questo è l'elenco delle cartelle e file che ritengo fondamentali per costruire l'applicazione.

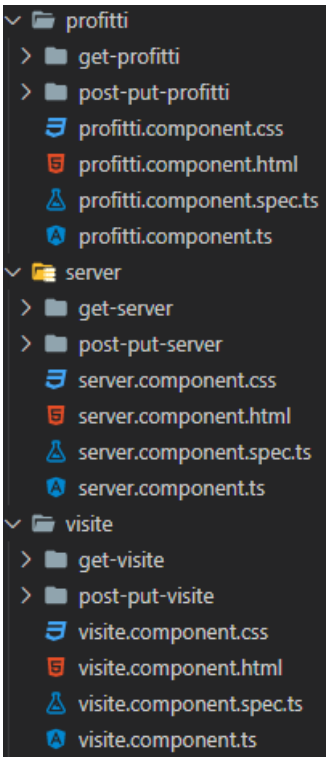
1. node_modules è stato già introdotto

2. src e app sono le cartelle che contengono tutta la parte di codice

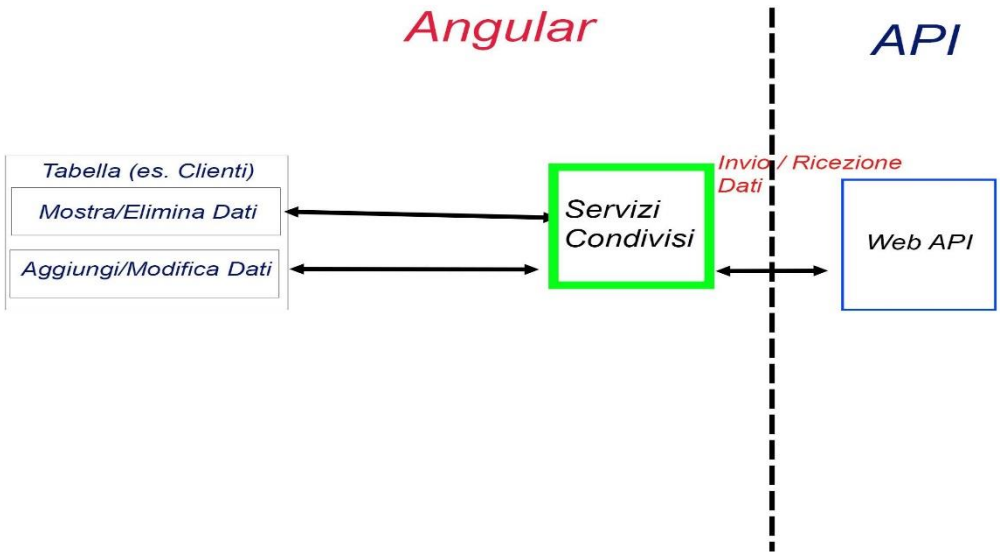
3. tutte le cartelle da clienti a visite le ho create con i comandi citati nel capitolo 4.

4. tutti i file app sono i primi presi in considerazione in fase di compilazione, e contengono tutti i componenti e moduli degli elementi dell'applicazione.

5. servizi.service.ts e dati.ts li descriverò dettagliatamente in seguito



Ogni componente che ha il nome uguale ad una tabella del database ha al suo interno altri due componenti, denominati get-componente e post-put-componente.

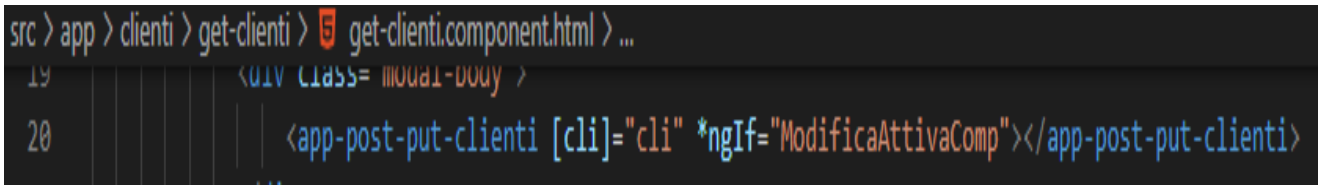


I componenti get servono per ricevere i dati dalla tabella del database, crearne una in html, con i tag `<table></table>`, `<tr></tr>`, `<td></td>` ed altre funzioni di Angular, come ad esempio *ngIf, una condizione di Angular che si può applicare al linguaggio HTML, oppure *ngFor che ha la stessa funzione di forEach, che ho usato per ricevere i dati dal database.

Ogni componente ha il proprio tag HTML, ad esempio get-clienti ha `<app-get-clienti></app-get-clienti>` ed esso è visibile nel codice Typescript del componente, dichiarato nel selettore @Component

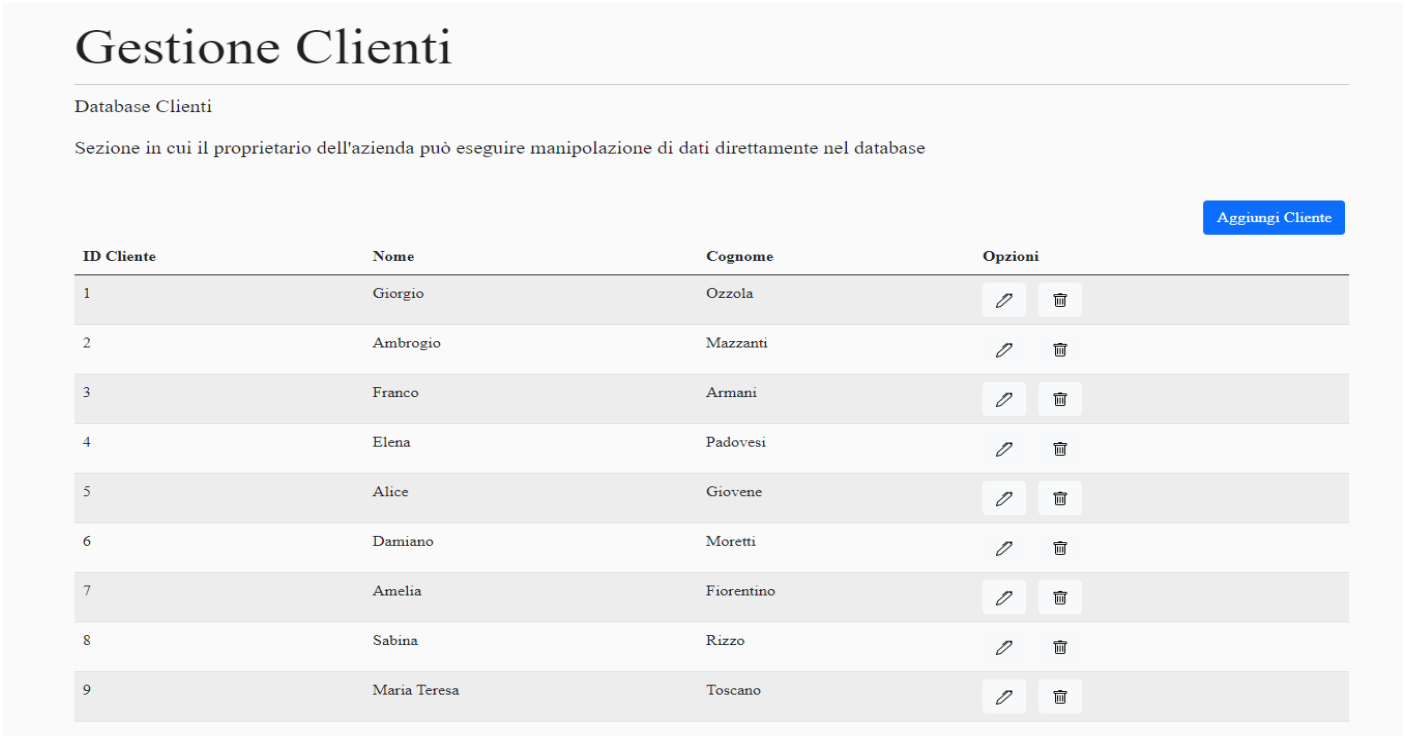
```
import { Component, OnInit } from '@angular/core';
import { ServiziService } from 'src/app/servizi.service';
@Component({
  selector: 'app-get-clienti',
  templateUrl: './get-clienti.component.html',
  styleUrls: ['./get-clienti.component.css'],
})
```

In questo componente si trova il tag HTML del componente post put, perché con get si riceve la tabella, mentre con il componente post-put si ha un modal[8] in cui si ha la possibilità di aggiornare oppure inserire un record nella tabella, che sarà visibile in tempo reale nel database.



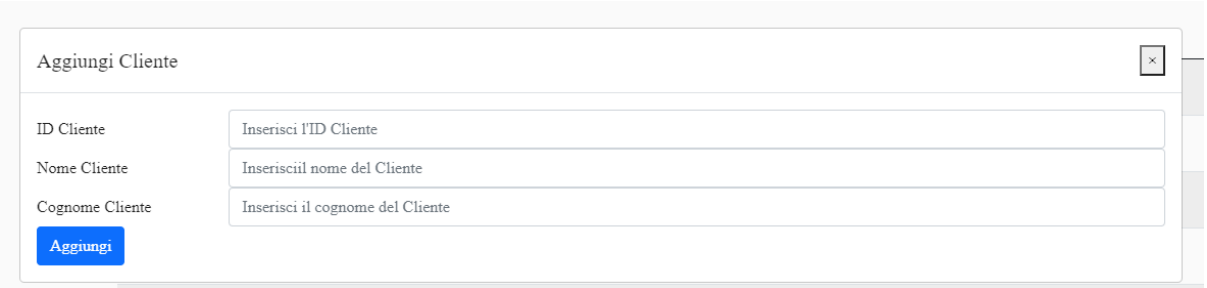
[8] "Finestra di dialogo - Wikipedia." https://it.wikipedia.org/wiki/Finestra_di_dialogo.

Questa è la tabella nell'interfaccia rispecchiante la tabella Clienti nel database:

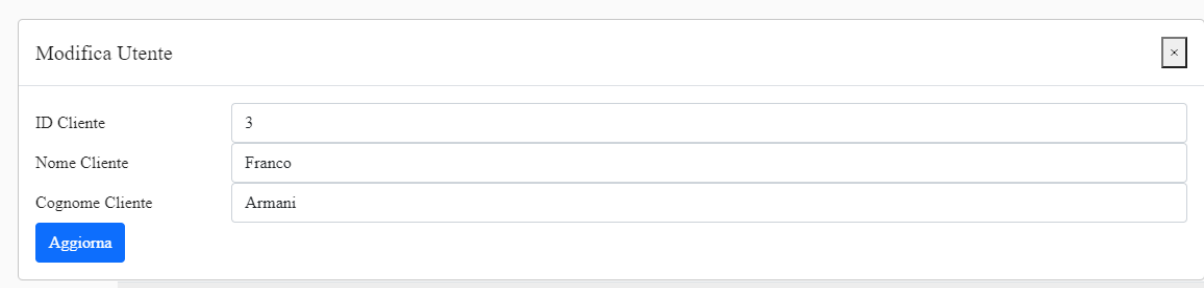


Il componente post-put viene richiamato dai due bottoni delle opzioni (la matita e il cestino) e dal bottone di aggiunta (quello blu)

Modal che si attiva al click del bottone Aggiungi Cliente



Mentre questo è il modal con già contente i dati che l'utente vuole modificare



Queste operazioni sono simili nelle pagine riferite alle altre tabelle, cambiano le classi a cui si riferiscono (per coincidere con l'API) e i metodi a cui fanno riferimento.
Ogni "click" dell'utente è associato ad una funzione nel file .ts del componente, che attiva un secondo metodo, questa volta HTTP, dichiarato nel file servizi.service.ts e questi sono gestiti dal componente get

```
export class GetClientiComponent implements OnInit {
  constructor(private servizio: ServiziService) {}
  ListaClienti: any = [];
  TitoloModal: string = '';
  cli: any;
  ModificaAttivaComp: boolean = false;
  ngOnInit() {
    this.AggiornaListaClienti();
  }

  aggiungiClick() {
    this.cli = {
      IdCliente: '',
      Nome: '',
      Cognome: '',
    };
    this.TitoloModal = 'Aggiungi Cliente';
    this.ModificaAttivaComp = true;
  }
  modificaClick(item: any) {
    this.cli = item;
    this.TitoloModal = 'Modifica Utente';
    this.ModificaAttivaComp = true;
  }
  eliminaClick(item: any) {
    if (confirm('Eliminare definitivamente questo cliente?')) {
      this.servizio.DELETE_Clienti(item.IdCliente).subscribe((data) => {
        alert(data.toString());
        this.AggiornaListaClienti();
      });
    }
  }
  chiudiClick() {
    this.ModificaAttivaComp = false;
    this.AggiornaListaClienti();
  }
  AggiornaListaClienti() {
    //subscribe si assicura di aver ricevuto tutti i dati prima di passare all'istruzione successiva
    this.servizio.GET_Clienti().subscribe((data) => {
      this.ListaClienti = data;
    });
  }
}
```

A sinistra: i metodi dei click in get-clienti.ts

Sotto si trovano i metodi riferiti alla pagina costi :

```
//#region Costi
GET_Costi(): Observable<any[]> {
  return this.http.get<any>(this.APIUrl + '/costi');
}
POST_Costi(val: any) {
  return this.http.post(this.APIUrl + '/costi', val);
}
PUT_Costi(val: any) {
  return this.http.put(this.APIUrl + '/costi', val);
}
DELETE_Costi(val: any) {
  return this.http.delete(this.APIUrl + '/costi' + '/' + val);
}
//#endregion
```

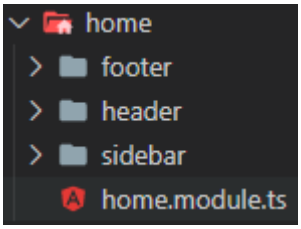
Per quanto riguarda invece il componente put-post.ts si dedica solamente all'aggiunta e all'aggiornamento di record da parte dell'utente (con record si intende elementi in tabella, ad esempio l'inserimento di un nuovo cliente).

In questa immagine si può osservare nel costruttore si riceve il servizio con i metodi http, che servono alle due funzioni aggiungiCliente() e aggiornaCliente() e come indicato dopo la prima riga, è il codice TypeScript del componente post-put-clienti.

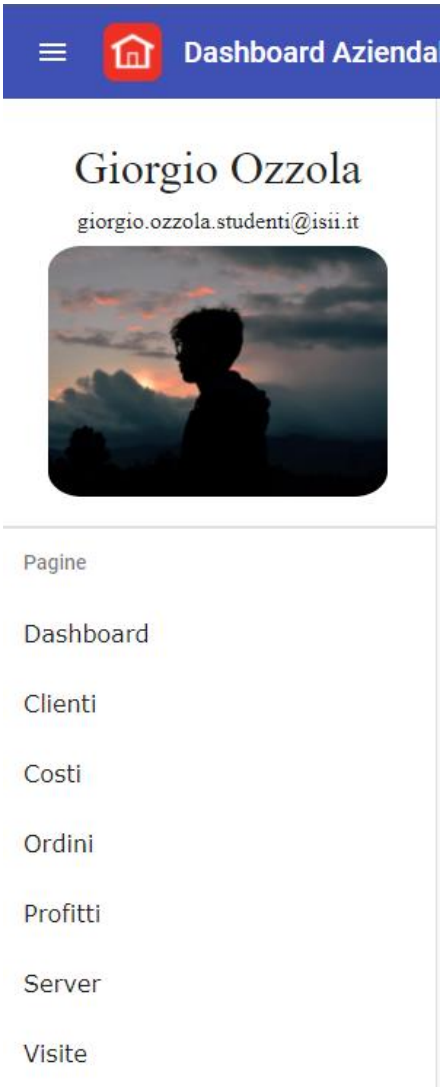
```typescript
export class PostPutClientiComponent implements OnInit {
  constructor(private servizio: ServiziService) {}

  @Input() cli: any;
  IdCliente: number = 0;
  Nome: string = '';
  Cognome: string = '';
  ngOnInit(): void {
    this.IdCliente = this.cli.IdCliente;
    this.Nome = this.cli.Nome;
    this.Cognome = this.cli.Cognome;
  }
  aggiungiCliente() {
    var val = {
      IdCliente: this.IdCliente,
      Nome: this.Nome,
      Cognome: this.Cognome,
    };
    this.servizio.POST_Clienti(val).subscribe(res => {
      alert(res.toString());
    });
  }
  aggiornaCliente() {
    var val = {
      IdCliente: this.IdCliente,
      Nome: this.Nome,
      Cognome: this.Cognome,
    };
    this.servizio.PUT_Clienti(val).subscribe(res => {
      alert(res.toString());
    });
  }
}
```

Capitolo 7 - Home, Default e Routing



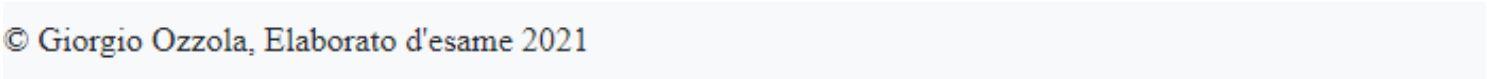
Riferito alla prima immagine del capitolo 5, vorrei specificare delle caratteristiche che possiedono questi due componenti.
Il primo componente: Home, a sua volta contiene altri tre componenti, header, sidebar e footer, ed un modulo, home.module.ts.
Questi componenti devono essere visualizzati durante la navigazione di tutte le pagine, tranne quella di login, che è la prima pagina all'entrata del sito.

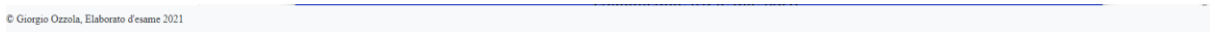


In alto :
il componente footer, con i bottoni Dashboard, Home e Profilo

A sinistra:
il componente side bar

In basso:
il componente footer





```
@NgModule({
  declarations: [
    HeaderComponent,
    FooterComponent,
    SidebarComponent,
  ],
  imports: [
    CommonModule,
    MatDividerModule,
    MatToolbarModule,
    MatIconModule,
    MatButtonModule,
    FlexLayoutModule,
    MatMenuModule,
    MatListModule,
    RouterModule,
  ],
  exports: [
    HeaderComponent,
    FooterComponent,
    SidebarComponent,
  ]
})
```

Questi componenti sono “contenuti” nel file home.module.ts, che li importa e li esporta, per caricarli nel componente Default

Default non solo è il primo componente che viene caricato, come indicato nel file app-routing-module.ts, ma gestisce anche il click nell’header per il componente side bar, così da potersi nascondere e mostrare con il click del bottone.
Nel suo modulo (default.module.ts) importa il modulo di Home (home.module.ts)

```
@NgModule({
  declarations: [
    DefaultComponent,
  ],
  imports: [
    CommonModule,
    RouterModule,
    HomeModule,
    MatSidenavModule,
    MatDividerModule,
    MatFormFieldModule,
    MatInputModule,
    MatCardModule,
    MatButtonModule,

  ]
})
```

Module Completo

Il “contenitore” di tutta l’applicazione lo fa il file app.module.ts, che in automatico, appena creato un nuovo componente, modulo o servizio li dichiara o importa i moduli.

```
@NgModule({
  declarations: [
    AppComponent,
    ClientiComponent,
    GetClientiComponent,
    PostPutClientiComponent,
    CostiComponent,
    GetCostiComponent,
    PostPutCostiComponent,
    OrdiniComponent,
    GetOrdiniComponent,
    PostPutOrdiniComponent,
    ProfittiComponent,
    GetProfittiComponent,
    PostPutProfittiComponent,
    ServerComponent,
    GetServerComponent,
    PostPutServerComponent,
    VisiteComponent,
    GetVisiteComponent,
    PostPutVisiteComponent,
    LoginComponent,
    VisitechartComponent,
    ServerchartComponent,
    ProfittichartComponent,
    OrdinichartComponent,
    Ordinichart2Component,
    CostichartComponent,
    ClientichartComponent,
    HomepageComponent,
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    HttpClientModule,
    FormsModule,
    ReactiveFormsModule,
    DefaultModule,
    BrowserAnimationsModule,
    ChartsModule,
    MatGridListModule,
    MatCardModule,
    MatMenuModule,
    MatIconModule,
    MatButtonModule,
    LayoutModule,
    NgxChartsModule,
  ],
  schemas:[CUSTOM_ELEMENTS_SCHEMA],
  providers: [ServiziService],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

```
import { CUSTOM_ELEMENTS_SCHEMA, NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { ClientiComponent } from './clienti/clienti.component';
import { GetClientiComponent } from './clienti/get-clienti/get-clienti.component';
import { PostPutClientiComponent } from './clienti/post-put-clienti/post-put-clienti.component';
import { ServiziService } from './servizi.service';
import { HttpClientModule } from '@angular/common/http';
import { FormsModule, ReactiveFormsModule } from '@angular/forms';
import { DefaultModule } from './default/default.module';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';
import { CostiComponent } from './costi/costi.component';
import { GetCostiComponent } from './costi/get-costi/get-costi.component';
import { PostPutCostiComponent } from './costi/post-put-costi/post-put-costi.component';
import { OrdiniComponent } from './ordini/ordini.component';
import { GetOrdiniComponent } from './ordini/get-ordini/get-ordini.component';
import { PostPutOrdiniComponent } from './ordini/post-put-ordini/post-put-ordini.component';
import { ProfittiComponent } from './profitti/profitti.component';
import { GetProfittiComponent } from './profitti/get-profitti/get-profitti.component';
import { PostPutProfittiComponent } from './profitti/post-put-profitti/post-put-profitti.component';
import { ServerComponent } from './server/server.component';
import { GetServerComponent } from './server/get-server/get-server.component';
import { PostPutServerComponent } from './server/post-put-server/post-put-server.component';
import { VisiteComponent } from './visite/visite.component';
import { GetVisiteComponent } from './visite/get-visite/get-visite.component';
import { PostPutVisiteComponent } from './visite/post-put-visite/post-put-visite.component';
import { LoginComponent } from './login/login.component';
import { ChartsModule } from 'ng2-charts';
import { MatGridListModule } from '@angular/material/grid-list';
import { MatCardModule } from '@angular/material/card';
import { MatMenuModule } from '@angular/material/menu';
import { MatIconModule } from '@angular/material/icon';
import { MatButtonModule } from '@angular/material/button';
import { LayoutModule } from '@angular/cdk/layout';
import { VisitechartComponent } from './dash/visitechart/visitechart.component';
import { NgxChartsModule } from '@swimlane/ngx-charts';
import { ServerchartComponent } from './dash/serverchart/serverchart.component';
import { ProfittichartComponent } from './dash/profittichart/profittichart.component';
import { OrdinichartComponent } from './dash/ordinichart/ordinichart.component';
import { Ordinichart2Component } from './dash/ordinichart2/ordinichart2.component';
import { CostichartComponent } from './dash/costichart/costichart.component';
import { ClientichartComponent } from './dash/clientichart/clientichart.component';
import { HomepageComponent } from './homepage/homepage.component';
```

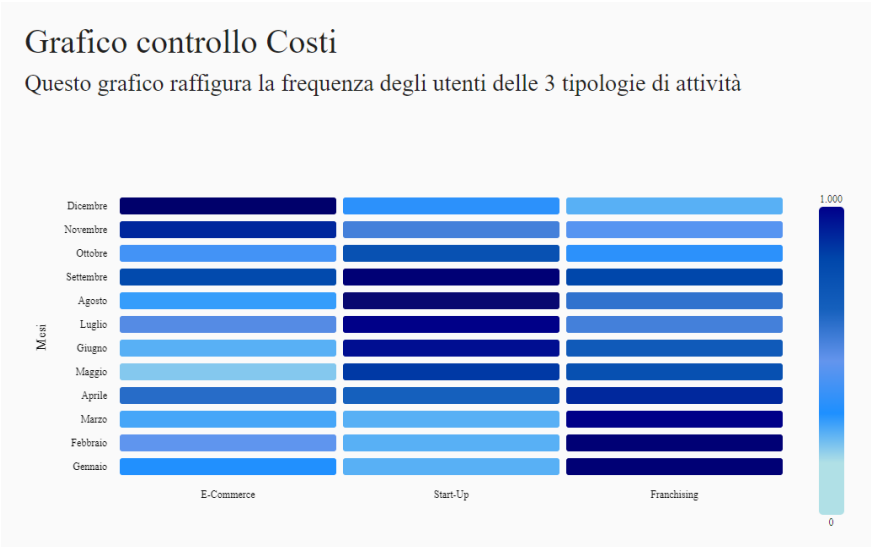
Routing

Come menzionato in precedenza, il file app-routing.module.ts è il file che gestisce tutti i path alle pagine e quale componente visualizzare a seconda dell’URL, senza ciò non si potrebbe navigare nelle pagine del sito. Degna di nota è la proprietà children, che fa visualizzare i componenti di Default sopracitati, con una sorta di “dipendenza”.

```
const routes: Routes = [
  {
    path: '', component: DefaultComponent,
    children: [
      {
        path: 'dash',
        component: DashComponent
      },
      {
        path: '',
        component: HomepageComponent
      },
      {
        path: 'piechart', component: VisitechartComponent
      },
```

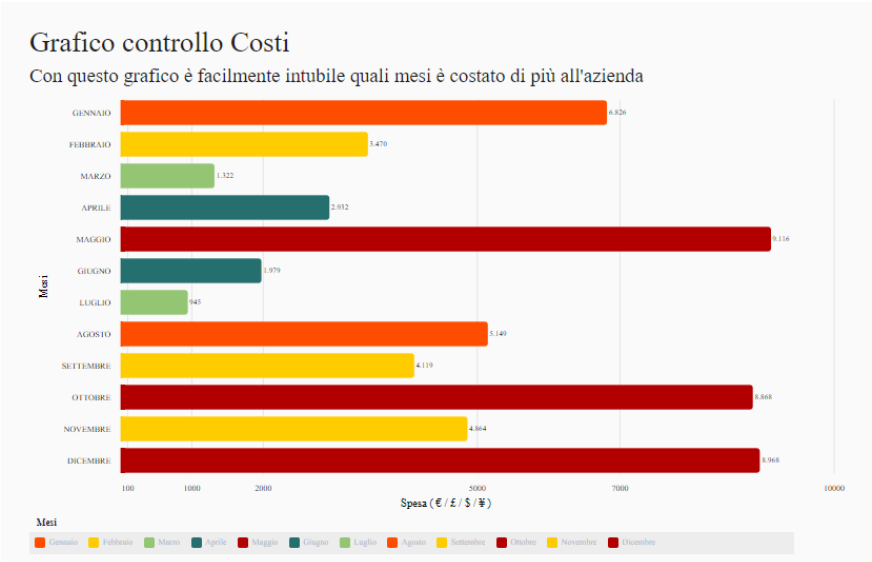
Capitolo 8 - Dashboard

Questo è il “core” dell’applicazione, cioè la gestione aziendale, con sei grafici, uno per tabella del database (per sottolineare la base di dati, fondamentale nello svolgimento del progetto). Per l’integrazione di questi grafici ho usato la libreria open-source Chart.js, per questo ho creato un file denominato dati.ts, in cui esporta i dati in un formato JSON.




```
export var DatiClienti = [
  {
    "name": "E-Commerce",
    "series": [
      {
        "name": "Gennaio",
        "value": 174
      }, {
        "name": "Febbraio",
        "value": 321
      }, {
        "name": "Marzo",
        "value": 121
      }, {
        "name": "Aprile",
        "value": 463
      }, {
        "name": "Maggio",
        "value": 50
      }, {
        "name": "Giugno",
        "value": 100
      }, {
        "name": "Luglio",
        "value": 364
      }, {
```

In alto: grafico di controllo Clienti
A destra: esempio di dato in dati.ts



Sopra: grafico di controllo dei clienti dell’azienda

Capitolo 9 - Socket (Porte)

Ho deciso di fare un capitolo sulle porte perché ritengo importante sottolineare l'importanza che un singolo indirizzo.
L'indirizzo di cui sto parlando è http://localhost , riprendendo ciò descritto nel capitolo 4, con il comando ng serve, viene creato un web server locale all'indirizzo predefinito http://localhost:4200, che è l'ambiente di sviluppo di Angular, mentre l'API usa http://localhost:5000, essendo dello stesso indirizzo ma su due porte diverse.
Infatti, nel codice TypeScript del servizio servizio.service.ts è presente l'url delle API per utilizzarle.

```
export class ServiziService {
  //passo gli URL come readonly e creo con questi i metodi per la pagina
  readonly APIUrl = 'http://localhost:5000/api';
  readonly FotoUrl = 'http://localhost:5000/salvafile';
  constructor(private http: HttpClient) { }
```

In questa immagine è presente la variabile readonly FotoUrl, per cui ne parlerò nel prossimo punto del documento

Conclusioni

Sono soddisfatto di tutto il lavoro che ho svolto in questo lasso di tempo, nonostante le complicazioni e il tempo che deve sempre essere considerato.
Ho rispettato in pieno il tema che avevo dichiarato nel mese di aprile, senza andare fuori contesto.
Ci sono diversi aspetti da "implementazione futura", uno di questi è un'implementazione che comprendeva l'inserimento di immagini profilo di ogni cliente, ma per il tempo non sono riuscito a portare a termine.
Un'altra implementazione riguarda i grafici di monitoraggio:
il file dati.ts contiene dei dati statici, quindi non importa che modifiche si attuino al database, i grafici rimarranno sempre uguali.
Bisogna evidenziare il fatto che questo progetto è in fase di sviluppo, quindi non è applicabile a nessuna azienda essendo una "demo".
L'ultima implementazione che inserirei è l'esportazione dei dati, perché se nel database sono presenti tutti i dati del sito aziendale del mese di maggio bisogna poterli esportare per comodità e assicurarsi un "backup" locale.

```
[Route("salvafile")]    //parametro dell'url
[HttpPost]
0 riferimenti
public JsonResult SalvaFile()
{
    try
    {
        var httpRequest = Request.Form;
        var postFile = httpRequest.Files[0];    //solo il primo file nel body della richiesta
        string nomeFile = postFile.FileName;
        string path = envirionment.ContentRootPath + "/Foto/" + nomeFile;
        using (var file = new FileStream(path, FileMode.Create))
        {
            postFile.CopyTo(file);
        }
        return new JsonResult(nomeFile);    //se vedo ritornare il nome del file allora è stato caricato
    }
    catch (Exception)
    {
        return new JsonResult("anonimo.png");   //per gestire l'eccezione comunque una foto ritorna
    }
}
```

Metodo funzionante per le foto

Bibliografia

Argomento	Autore	URL	Tipo
@Output ed Eventmitter	iFour Technolab	https://www.ifourtechnolab.com/blog/understanding-output-and-eventemitter-in-angular	Blog
ASP.NET Core	Microsoft Docs	https://docs.microsoft.com/it-it/aspnet/core/mvc/overview?view=aspnetcore-5.0	Forum
Angular.io	Angular Official Site	https://angular.io/guide/setup-local	Forum
Bootstrap	Bootstrap.com	https://getbootstrap.com	Pagina Web
Schema MVC	Microsoft Docs	https://docs.microsoft.com/it-it/aspnet/core/mvc/overview?view=aspnetcore-5.0	Forum
Sidenav API	Angular API	https://material.angular.io/components/sidenav/api	Forum
StackBlits	StackBlitz	https://stackblitz.com	Pagina Web

Materiale utilizzato

Elenco applicativi utilizzati in questo progetto

• Visual Studio 2019	IDE sviluppato da Microsoft
• Angular Framework	Framework
• SQL Server	Sistema Operativo lato server
• Node-JS	Runtime Javascript
• Notion	Applicazione per la produttività
• Visual Studio Code	IDE sviluppato da Microsoft
• SMSS	IDE sviluppato da Microsoft per il lato server
• Hyper-V	Creatore di macchine virtuali
• Post man	Programma per testare URL