UNIVERSITÀ
DEGLI STUDI
DI PADOVA

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

Corso di laurea in Ingegneria informatica

Dipartimento di ingegneria dell'informazione

Progettazione e sviluppo di una applicazione web gestionale per la manutenzione di apparati industriali

Relatore:
Carlo Fantozzi

Laureando:
Hermann Serain

Anno accademico 2023/2024
Data di laurea 24/09/2024

Sommario

L'obiettivo di questa relazione sull'attività lavorativa è trattare i concetti fondamentali di ASP.NET Core alternando spiegazioni teoriche ad esempi pratici, per progettare e sviluppare un'applicazione di manutenzione di apparati industriali. Verranno esplorati diversi aspetti chiave della piattaforma, iniziando con i concetti fondamentali della tecnologia utilizzata, come il meccanismo di compilazione ed esecuzione, che stanno alla base per il funzionamento della piattaforma. Successivamente verranno esaminati i Middleware, strumenti cruciali per la gestione delle richieste HTTP che grazie alla costruzione di una pipeline strutturata permettono di implementare numerosi controlli e funzionalità.
Un'attenzione particolare sarà dedicata alla definizione di un servizio e al suo utilizzo tramite la Dependency Injection, un principio cardine di ASP.NET Core che permette di rendere disponibile in tutto il progetto diverse funzionalità in base alle esigenze. La relazione discuterà anche del routing delle pagine HTML e della gestione della logica di funzionamento sia lato server (con Razor Pages) che lato client (alcuni accenni sui file JavaScript e le tecnologie utilizzate).
La manipolazione dei dati è un aspetto fondamentale per qualsiasi applicazione software: questa avviene attraverso Entity Framework Core, una tecnologia che offre funzionalità avanzate per l'interrogazione, l'inserimento e la modifica dei dati. Un altro argomento di grande importanza nello sviluppo di web app è l'autenticazione e autorizzazione, implementata in ASP.NET Core tramite il framework "Identity".
Vorrei fare una breve premessa sull'uso del materiale del corso **ASP.NET Core per tutti** [1]. Questo materiale proviene da un corso di ASP.NET Core che ho seguito nel 2020 per prepararmi al lavoro in azienda. Ho selezionato le immagini principali per spiegare la teoria, mentre i concetti presentati sono stati trattati in base alle mie conoscenze.

Indice

1 Introduzione

In questo capitolo viene brevemente presentata l'azienda di riferimento, l'applicazione per la manutenzione di apparati industriali e l'obbiettivo dell'attività lavorativa.

1.1 Omniacore Solutions

Omniacore Solutions [2] è l'azienda in cui lavoro e che mi ha permesso di svolgere le 225 ore (circa) di attività lavorativa su cui si basa la seguente relazione. L'obiettivo di questa azienda è proporre soluzioni software (e hardware) innovative utilizzando le migliori tecnologie disponibili sul mercato, come ad esempio:

- **Mobile**: Web app native, Java, Flutter, Kotlin, Ionic
- **Desktop**: VB.net, C#
- **Database**: SQL Server, MySQL, MongoDB, PostgreSQL
- **Back-end**: PHP, Node.js, Python, ASP.Net Core
- **Front-end**: HTML, CSS, TypeScript, JavaScript, jQuery

Alcuni progetti di riferimento includono:

- **Upndw** [3], un social network per il mondo finanziario e del trading online, che si distingue dagli altri social grazie a funzionalità specifiche come la ricezione delle quotazioni in tempo reale e i grafici avanzati sviluppati con librerie grafiche.
- **Pubblie** [4], una piattaforma che consente di gestire facilmente più social network contemporaneamente, l'ideale per social media manager o per chi desidera pubblicizzare la propria attività.
- **SCADA**, una soluzione per l'industria 4.0 che permette di monitorare e intervenire sulle macchine in tempo reale in modo da tenerle sotto controllo in base ai parametri di funzionamento.

Il tratto distintivo di Omniacore è la passione e professionalità che i componenti dell'azienda hanno nel proprio lavoro; ciò consente ai clienti di usufruire di un supporto completo per lo sviluppo di progetti in base ai loro desideri e necessità.

1.2 Panoramica sul progetto

Il progetto in cui si inserisce la mia attività nasce da una specifica esigenza: centralizzare la gestione della manutenzione degli apparati industriali, in particolare dei bruciatori industriali. L'azienda che ha richiesto la collaborazione si chiama **Sartorello Bruciatori** [33], lavora a stretto contatto con **Omniacore Solutions** ed è specializzata nella manutenzione e installazione di bruciatori industriali. La struttura organizzativa prevede la gestione dei clienti attraverso una lista classica inserita in un database contabile, che include le sedi dei vari clienti e, per ogni sede, i macchinari associati. Lo stesso software gestisce anche i contratti di lavoro stipulati con i clienti, specificando le tempistiche per gli interventi di manutenzione ordinaria e i parametri per la manutenzione straordinaria.
Gli interventi straordinari avvengono nel caso ci siano delle emergenze improvvise o delle chiamate di intervento che non rispettano i contratti di manutenzione. Gli interventi ordinari, invece, riguardano operazioni di controllo e manutenzione previste dai contratti (ad esempio, la pulizia di singoli macchinari oppure di interi impianti), solitamente questa tipologia di lavoro avviene in modo periodico, per esempio con cadenza mensile, trimestrale, quadrimestrale o annuale. Alla base di questi interventi ci sono gli operatori che svolgono tale lavoro, questi professionisti prendono il nome di manutentori. I manutentori lavorano in squadre e ogni squadra può esser composta da tecnici di diverso grado e categoria. I maggiori utilizzatori del software di manutenzioni sono proprio loro, visto che dovranno andare a compilare un rapportino di lavoro finale per poter fare un riassunto del lavoro svolto

durante la giornata, operazione che in precedenza veniva svolta con moduli cartacei. La gestione degli interventi viene effettuata tramite un software separato, in questo caso un semplice foglio Excel, che consente di monitorare i giorni di lavoro e le squadre disponibili. Questo strumento permette di organizzare efficacemente il lavoro delle varie squadre, assegnando loro le diverse sedi e compiti da svolgere in base alle disponibilità. Infine, altre funzionalità, come la gestione dei pezzi di ricambio, il salvataggio di foto che documentano lo stato dei macchinari, e la produzione di documenti legali per la certificazione dello stato degli impianti, viene gestite con software diversi o, in alcuni casi, con l'uso di documenti cartacei.
Per risolvere questa gestione eterogenea dell'attività lavorativa si è deciso di sviluppare un'applicazione che ha i seguenti **requisiti principali**: gestione anagrafica dei clienti con relative sedi e macchinari, gestione dei contratti di lavoro e degli interventi (sia ordinari che straordinari), gestione della tipologia di squadre che devono fare un tipo di intervento, creazione assistita di documenti burocratici, possibilità di creare dei rapportini di fine lavoro da parte dei manutentori e definizione di una parte dedicata al magazzino per la gestione assistita dei pezzi. Queste funzionalità rese disponibili da parte dell'applicazione non saranno visibili a tutta l'utenza, ma solo ad una parte in base al ruolo che possiedono. Precedentemente è stato menzionato il ruolo del manutentore ma, oltre ai manutentori, come ruoli utente del progetto sono stati previsti: degli utenti amministratori, ovvero noi di Omniacore, che possiedono tutti i privilegi; degli utenti manager che si occupano di gestire i ruoli degli utenti, le parti più sensibili della anagrafica e avviare un nuovi intervento. Infine gli ultimi due ruoli sono quello di "operatore" per dare la possibilità ai clienti di vedere alcune parti riguardanti la loro anagrafica e quello "guest" che corrisponde all'utente senza permessi.
Lo sviluppo di questo software ha permesso di velocizzare diverse operazioni, come la produzione e stampa autonoma di documenti basati sui dati presenti nella piattaforma, l'aggiornamento del magazzino da parte dei manutentori con l'allegato di foto, la possibilità di fare report di lavoro in base al tipo di intervento svolto e molto altro ancora potrà essere sviluppato in base a future richieste di aggiornamento. Un concetto fondamentale che ha guidato lo sviluppo di questo progetto è stato il rendere il più generale possibile la piattaforma, per poterla utilizzare come programma di manutenzione generico e dunque non solo per i bruciatori industriali.

1.3 Obiettivi e organizzazione dell'attività lavorativa

L'obiettivo principale della attività lavorativa era progettare e sviluppare un programma da zero, basandosi sulle esperienze assimilate sia nel lavoro che nell'Università. Da un lato, si dovevano mettere in pratica tutte le nozioni e competenze acquisite nell'ambito della tecnologia (grazie al corso **ASP.NET Core per tutti** [1] e ad esperienze passate); dall'altro, ci si concentrava sulle tecniche di progettazione e ottimizzazione dei processi di lavoro grazie anche ai corsi svolti in merito (ad esempio ingegneria del software). Oltre a questo obiettivo personale, vi era l'obiettivo aziendale di colmare le lacune presenti, come descritto nel paragrafo precedente.
La prima parte del lavoro ha riguardato la progettazione degli elementi principali e le loro funzionalità associate, necessarie per soddisfare tutte le esigenze emerse. Dopo aver inizializzato le tabelle del database e sviluppato le operazioni fondamentali si è passati alla definizione delle interfacce web e delle loro funzionalità (la parte più corposa). Durante lo sviluppo dell'applicazione, sono state aggiunte funzionalità e miglioramenti vari, in base alle problematiche riscontrate o agli accorgimenti fatti in corso d'opera, senza però cambiare lo scheletro e le funzionalità di base del programma. Inoltre il flusso di lavoro che si è seguito è quello del metodo **agile** di base, descritto nella sezione 3.1.
Un'altra parte molto importante del programma riguarda la definizione dei ruoli e delle azioni possibili per ciascun utente registrato nella piattaforma. Ad esempio, nelle piattaforme classiche si trova sempre un utente admin con tutti i permessi, un utente manager con la possibilità di gestire gli utenti, un utente base che utilizza maggiormente il programma e infine un utente guest che non ha alcun permesso.

1.4 Asp.NET Core

L'utilizzo del framework ASP.NET Core per lo sviluppo di questa applicazione è stato scelto per diversi motivi, sia legati al linguaggio di programmazione e design pattern adottati, sia per modularità e funzionamento multi

piattaforma. Vediamo con precisione queste motivazioni di seguito:

- **Linguaggio C#**.
  Alla base dello sviluppo software adottato dall'azienda vi è la scelta di un linguaggio solido e moderno come C#. Questa scelta è stata guidata non tanto da un confronto con altri linguaggi, ma piuttosto dalla volontà di valorizzare i vantaggi intrinseci di C# in relazione all'uso di ASP.NET Core. C# si distingue per le sue caratteristiche concettuali e strutturali, come il robusto supporto alla programmazione orientata agli oggetti e la capacità di evolversi rispetto ai linguaggi precedenti, come Java. Inoltre, C# integra sia costrutti tradizionali, ereditati da C/C++, sia funzionalità moderne allineate con le tecnologie emergenti, come ad esempio la libreria **ML .NET legata all'AI** [9], che lo rendono particolarmente adatto per lo sviluppo di applicazioni ASP.NET Core.

- **Modularità**.
  L'organizzazione e la gestione delle risorse sono caratteristiche fondamentali per un'applicazione ordinata ed efficiente. Questo aspetto è cruciale sia per le integrazioni future (poiché un software disordinato risulta difficile da modificare dopo un lungo periodo di tempo) sia per l'ottimizzazione delle risorse (è possibile decidere quali componenti includere nella nostra applicazione, ottenendo così performance di funzionamento elevate). La gestione delle risorse diventa particolarmente rilevante anche quando si affrontano problematiche legate alla gestione di una grande mole di dati o alla concorrenza, dove molti processi (o thread) possono scrivere e leggere sulla stessa risorsa.
  Un esempio concreto di come implementare questa proprietà è l'utilizzo dei pacchetti **NuGet** [10], che consentono di integrare librerie e pacchetti esterni nella compilazione specificandoli nel file ***.csproj** (discusso nella sezione 3.2.1). Questo approccio permette di includere nel nostro progetto solo ciò che effettivamente utilizziamo ottimizzando l'utilizzo delle risorse, e esemplifica l'aspetto della modularità dove ogni componente software ha un compito ben preciso.

- **Cross-Platform**.
  Per sviluppare software utilizzato da una vasta gamma di utenti è fondamentale garantire che possa essere eseguito su più piattaforme. Esistono due principali approcci per raggiungere questo obiettivo: il primo consiste nello scrivere programmi in codice nativo per ogni piattaforma che si vuole coprire, mentre il secondo prevede lo sviluppo di software utilizzando una tecnologia multi piattaforma. Come si può intuire il primo approccio richiede molti più sforzi rispetto al secondo, sia in termini di risorse che di tempo. Inoltre, a meno che non siano necessarie funzionalità particolari risulta inefficiente sviluppare lo stesso software in codice nativo per ogni piattaforma.
  L'utilizzo di una tecnologia multi-piattaforma come ASP.NET Core consente di sviluppare applicazioni compatibili con diversi dispositivi e sistemi operativi, tra cui Windows, macOS e Linux. Questa versatilità si estende anche ai microprocessori basati su architetture ARM a 32 e 64 bit (ARM32/64). Per questo ultimo esempio ASP.NET Core risulta essere particolarmente interessante perché permette di creare applicazioni che funzionano su dispositivi come il Raspberry Pi, offrendo così la possibilità di sviluppare soluzioni innovative che spaziano oltre il semplice sviluppo di Web App, andando a creare delle vere e proprie applicazioni Embedded.

- **Open Source**.
  ASP.NET Core è una tecnologia **open source**, che viene sempre mantenuta da Microsoft ma a differenza di altri progetti vengono accettate integrazioni da altri sviluppatori esterni. Questa caratteristica principale comporta numerosi vantaggi descritti di seguito. Innanzitutto ogni sviluppatore ha accesso al codice sorgente, potendo così evidenziare potenziali problemi o aree di miglioramento, garantendo una maggiore trasparenza tra gli utilizzatori della tecnologia e gli sviluppatori e sostenendo una tecnologia aggiornata e innovativa.
  È importante notare che Microsoft mantiene il controllo del progetto per assicurare un prodotto di qualità agli utilizzatori, andando cosi a portare ad un ampio supporto in termini di documentazione e a costi significativamente inferiori rispetto alle tecnologie gestite in modo centralizzato.

In sintesi, una tecnologia **open source** sottoposta ai dovuti controlli risulta essere molto più efficace ed efficiente rispetto a una tecnologia centralizzata, dove tutte le fasi di progettazione, sviluppo e mantenimento dipendono da un'unica entità.

- **Funzionalità Integrate e performace**.
  Oltre alle proprietà riguardati la tecnologia in generale, ASP.NET Core offre diversi costrutti architetturali e metodologici molto interessanti e variegati. Per fare alcuni esempi, grazie al pacchetto **Identity** [6] è possibile implementare facilmente potenti meccanismi di autenticazione e autorizzazione per poter garantire la sicurezza della nostra applicazione e per poter fare in modo che ogni utenza abbia la sua corretta funzione. Dal punto di vista architetturale invece, c'è molta flessibilità per quanto riguarda i componenti che la costruiscono rendendo la tecnologia il più personalizzabile possibile; questo avviene grazie ad un **hosting flessibile** [7] che verrà anche accennato nella sezione 3.2.1. Infine per menzionare alcuni aspetti di programmazione, le ultime tecnologie hanno introdotto un modo efficace e funzionale per poter scrivere con il codice C# anche lato client, grazie alle **Blazor** [8] (ovvero naturali sostituti, o quasi, delle pagine JavaScript).

2 Strumenti

In questo capitolo vengono introdotti gli strumenti utilizzati per poter progettare, sviluppare e pubblicare un software utilizzando ASP.NET Core. Insieme agli aspetti generali vengono integrati alcuni esempi di utilizzo pratico per dare un risvolto concreto agli argomenti trattati.

2.1 Ambiente di Sviluppo Integrato

Il primo strumento software di cui vorrei parlare è l'IDE (Integrated Development Environment, in italiano Ambiente di Sviluppo Integrato). Questo software permette di velocizzare le operazioni di sviluppo fornendo solitamente un editor di codice sorgente, strumenti di automazione della build, un debugger e molte altre funzionalità. Come gli scrittori usano gli editor di testo e i contabili i fogli di calcolo, gli sviluppatori di software usano gli IDE per rendere il proprio lavoro più facile.
Sebbene sia possibile utilizzare qualsiasi editor di testo per scrivere codice, la maggior parte degli IDE offre funzionalità che vanno oltre la semplice modifica del testo. Essi forniscono un'interfaccia unificata per strumenti comuni agli sviluppatori, rendendo il processo di sviluppo del software molto più efficiente. Gli sviluppatori possono iniziare a programmare nuove applicazioni rapidamente, senza dover integrare e configurare manualmente software differenti e senza dover conoscere tutti gli strumenti individualmente, poiché possono concentrarsi su una sola applicazione. Di seguito sono elencate alcune ragioni per cui gli sviluppatori utilizzano gli IDE:

1. **Supporto multi-linguaggio**
   Lavorare con diverse tecnologie richieste dal mercato implica l'utilizzo di vari linguaggi di programmazione. Avere un IDE che supporta un'ampia gamma di linguaggi di programmazione è la soluzione ideale per uno sviluppatore, poiché consente di evitare la necessità di imparare a utilizzare molteplici ambienti di sviluppo.

2. **Editing del codice automatizzato**
   I linguaggi di programmazione hanno regole specifiche su come le istruzioni devono essere strutturate. Poiché l'IDE conosce queste regole, offre numerose funzionalità intelligenti per scrivere e modificare automaticamente il codice sorgente. Questo aspetto risulta molto utile sia per gli sviluppatori che sono alle prime armi e che dunque vengono guidati dall'IDE nel comprendere i costrutti fondamentali, sia per gli sviluppatori senior che grazie a queste funzionalità velocizzano il classico processo di sviluppo. Enunciamo di seguito alcuni esempi principali:

   - **Evidenziazione della sintassi**
     Un IDE può formattare il testo scritto utilizzando caratteri in grassetto o corsivo per alcune parole, applicando delle indentazioni alle nuove istruzioni oppure impiegando diversi colori per i font. Queste

indicazioni visive rendono il codice sorgente più leggibile e forniscono feedback immediato sugli errori sintattici accidentali. Quando verranno fornite varie immagini di codice, questa caratteristica risulterà immediatamente evidente.

- **Completamento intelligente del codice**
  Vari termini di ricerca appaiono quando si inizia a digitare una parola in un motore di ricerca. Allo stesso modo, un IDE può dare suggerimenti per completare un'istruzione del codice quando lo sviluppatore comincia a digitare.
- **Automazione dello sviluppo locale**
  Gli IDE migliorano la produttività dei programmatori eseguendo task di sviluppo ripetitivi, che sono solitamente parte di ogni modifica del codice. Ad esempio, offrono funzionalità di refactoring del codice che permettono di rinominare facilmente una classe e tutti i suoi riferimenti all'interno del progetto. Un'altra funzionalità interessante è la possibilità di definire costrutti di codice ben definiti, come classi, proprietà, costrutti di controllo (if, else, switch, for, ecc.), con un numero ridotto di clic.

3. **Compilazione**
   Un IDE compila e converte il codice in un linguaggio che il sistema operativo può comprendere. Alcuni linguaggi di programmazione implementano la compilazione just-in-time, in cui l'IDE converte il codice leggibile dall'umano in codice macchina direttamente all'interno dell'applicazione. Questa fase verrà spiegata più dettagliatamente nella sezione 3.2.1.1, dove si parlerà della compilazione di ASP.NET Core. Grazie a un buon IDE, sarà possibile applicare queste procedure con un singolo passaggio.

4. **Testing**
   La fase di testing è molto importante nel processo di produzione di un software e renderla il più automatica possibile porta ad un aumento dell'efficienza del programma e un uso ridotto di risorse (sia di tempo che di persone). L'IDE permette agli sviluppatori di automatizzare i test unitari localmente, prima che il software venga integrato con altri codici di altri sviluppatori e che si eseguano test di integrazione più complessi.

5. **Debugging**
   Il debugging è il processo di riparazione di qualsiasi errore o bug che il testing rivela. Una delle caratteristiche più importanti di un IDE ai fini del debugging è la possibilità di scorrere il codice, riga per riga, man mano che viene eseguito, e ispezionare il suo comportamento. L' IDE integra anche vari strumenti di debugging che evidenziano i bug causati dall'errore umano, in tempo reale, anche mentre lo sviluppatore sta digitando. Utilizzare un IDE con una solida gestione della fase di debugging può rendere più veloce ed efficiente lo sviluppo di codice.

6. **Collaborazione e Controllo del codice**
   Un altro aspetto fondamentale è il controllo del codice e di tutte le sue versioni. Il software che meglio realizza questa esigenza è **Git** [12] che verrà trattato nella prossima sezione. Avere un IDE con una buona integrazione con Git è sicuramente un fattore molto importante da tenere in considerazione, sia per poter fare un miglior refactoring del codice che per poter lavorare meglio in team sullo stesso progetto.

7. **Nuove tecnologie, funzionalità AI**
   Una tecnologia in forte sviluppo in questo momento è l'AI, che sta rivoluzionando il lavoro in diversi settori. Anche nello sviluppo di codice, l'AI può risultare estremamente utile se utilizzata in modo appropriato. Ad esempio, per svolgere operazioni ripetitive e monotone, l'AI si dimostra molto efficace ed efficiente. Per fare un esempio concreto, se si desidera creare una classe con lo scopo di deserializzare un grande oggetto JSON, l'AI potrebbe generare la classe con i campi e le proprietà corrette in pochi secondi. Inoltre, l'AI risulta molto utile anche nella creazione di enumeratori e switch di grandi dimensioni.

Oltre alle caratteristiche sopra elencate, è importante considerare le tipologie di IDE disponibili, che si suddividono in due principali categorie. La prima categoria comprende gli IDE classici locali, che consistono in un'applicazione desktop con la maggior parte delle funzionalità disponibili senza la necessità di una connessione internet. La

seconda categoria comprende gli IDE cloud, che possono essere utilizzati da qualsiasi browser e sfruttano risorse presenti su un'altra macchina.
Per motivi di prestazioni e usabilità, è stato scelto di utilizzare un IDE della tipologia classica. Inoltre, considerando la tecnologia utilizzata e le proprietà sopra menzionate, è stato selezionato come IDE di programmazione **Visual Studio** [10].

Visual Studio [10] è un IDE di programmazione molto comune per coloro che sviluppano nei linguaggi vicini all'ambiente Microsoft come C#, F#, C/C++ e VB, ma anche per altri linguaggi come JavaScript, TypeScript, Python e altri ancora. Questo ambiente di sviluppo offre un ampio spettro di funzionalità e interazioni con vari strumenti, che risultano in linea con le precedenti proprietà ricercate. L'aspetto principale che caratterizza Visual Studio rispetto ad altri IDE è la facilità con cui permette di tenere il codice ordinato grazie a potenti strumenti di navigazione, organizzazione, controllo del codice e debugging.
Di seguito è riportato un esempio di finestra di lavoro di **Visual Studio**, con le schede più interessanti e le relative funzionalità:

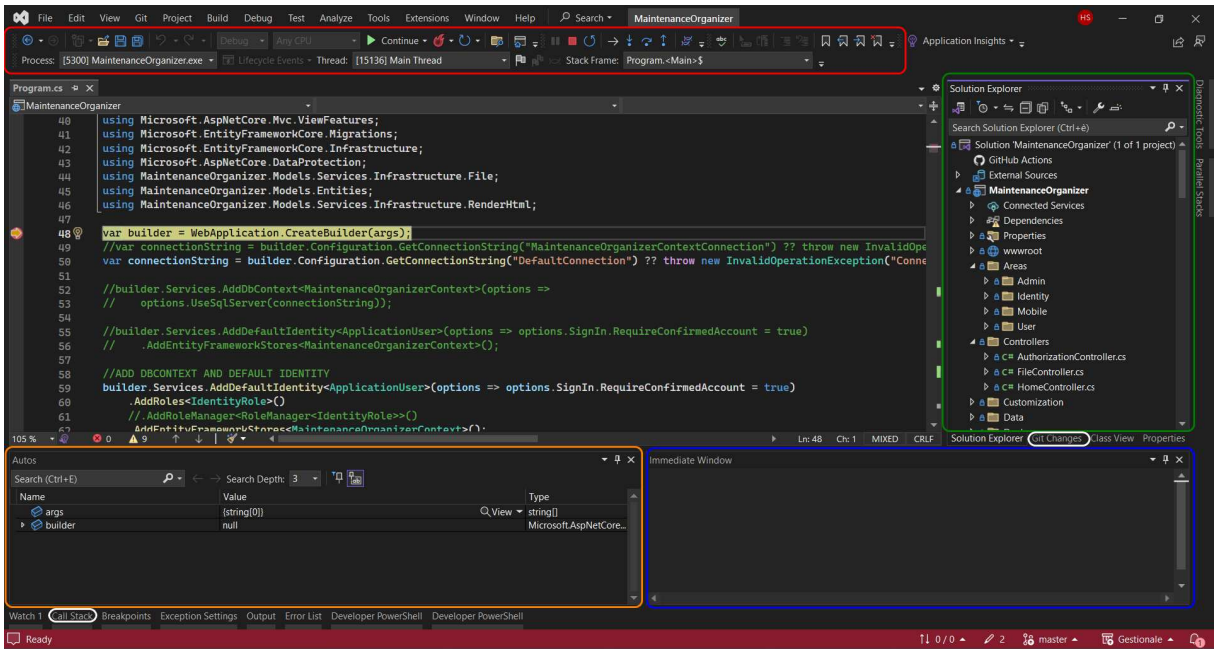


Fig. 1. Esempio di finestra di lavoro di Visual Studio.

Nella figura sono state evidenziate con dei colori le schede e le aree che verranno discusse in questo elenco:

- Sezione debug: In questa sezione sono presenti i classici pulsanti per il salvataggio, l'avvio dell'applicazione e la gestione dell'**Instruction Pointer** (la freccia gialla che indica la prossima istruzione da eseguire). Inoltre, sono visualizzate altre informazioni riguardanti il processo e il thread corrente che stanno eseguendo il codice. Infine, c'è una piccola sezione che indica in quale frame dello stack ci si trova durante l'esecuzione delle istruzioni correnti.
- Solution explorer: Classica finestra di esplora risorse che permette di cercare i file per nome e anche di riorganizzarne la vista. In parte ai vari file sono visibili dei lucchetti che stanno ad indicare l'avvenuto salvataggio su una cartella Git (sez. 2.2), inoltre è possibile vedere delle spunte rosse quando si va a fare una modifica ad un file o un più in verde quando si aggiunge un file nuovo.
- Autos: Finestra che permette di vedere i valori delle variabili correnti, cioè le variabili globali e quelle che si trovano nello scope della funzione che si sta analizzando. Questa finestra può venir utilizzata in coppia con la finestra **Watch** che permette di monitorare delle variabili aggiuntive rispetto a quelle presenti nella finestra Autos.

- Immediate Window: La Immediate Window in Visual Studio è una finestra di debug che permette di eseguire comandi, espressioni e funzioni direttamente durante l'esecuzione di un'applicazione. Si può utilizzare per valutare espressioni, visualizzare variabili, e chiamare metodi senza interrompere il flusso del programma, facilitando così il processo di debugging e test.
- **Call Stack**: Permette di vedere le istruzioni eseguite dal thread che si desidera esaminare. (Scheda indicata con un riquadro bianco in fondo all'immagine)
- **Git changes**: Finestra analizzata nelle sezioni successive che riguarda la gestione del codice lato Git/Github.

Queste finestre sono quelle principalmente utilizzate nelle realizzazione del progetto. Come si può intuire mettono a disposizione funzionalità molto comode e potenti che permettono di agevolare il processo di sviluppo.

2.2 Controllo Versione

Il secondo strumento software presentato nasce da una necessità ben precisa: la possibilità di monitorare le modifiche apportate a un file o a un insieme di file nel tempo, consentendo di richiamare versioni specifiche in seguito. Ad esempio, chi lavora in team su un progetto composto da numerosi documenti può avere la necessità di riportare alcuni file a uno stato precedente, ripristinare l'intero progetto a una versione passata, confrontare le modifiche nel tempo, vedere chi ha apportato l'ultima modifica che potrebbe aver causato un problema, capire chi ha introdotto un problema e quando, e molto altro ancora. Tutto questo è reso possibile grazie a un VCS, ovvero un Version Control System. Durante lo sviluppo e l'evoluzione di questa tecnologia, si è passati da implementazioni più semplici a quelle più complesse e strutturate, ciascuna con i propri vantaggi e svantaggi.
Esaminiamo alcune di queste architetture facendo riferimento alla **Fig. 2**.

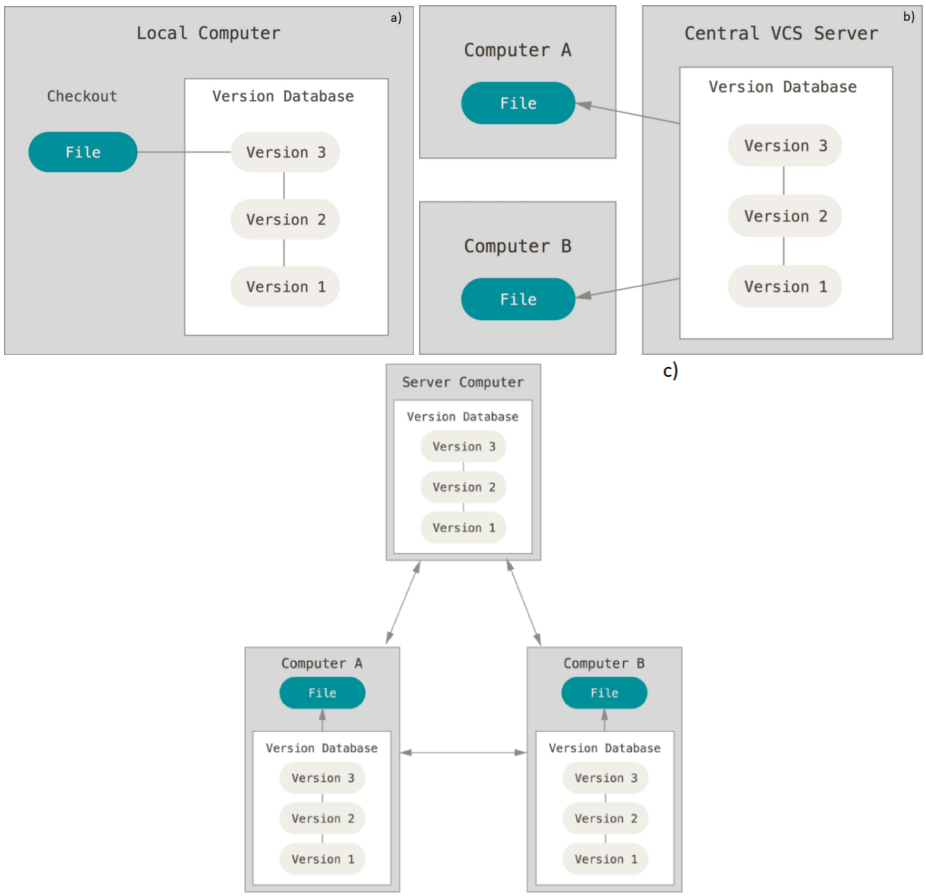


Fig. 2. Tipologie di VCS, fonte [12].

Il primo VCS è il **localized VCS** (**Fig. 2**, a). Il **localized VCS** è un metodo di controllo delle versioni che molte persone preferiscono, ovvero copiare i file in un'altra directory, magari una directory con un timestamp se sono previdenti. Questo approccio è molto comune per la sua semplicità, ma è anche estremamente soggetto a errori, infatti è facile dimenticare in quale directory ci si trova e finire per scrivere accidentalmente nel file sbagliato o copiare file indesiderati. Per affrontare questo problema, i programmatori hanno sviluppato da tempo VCS locali che utilizzano un semplice database per tenere traccia di tutte le modifiche fatte ai file sotto controllo di revisione. Uno degli strumenti VCS più popolari era un sistema chiamato **RCS** [12], che è ancora distribuito con molti computer oggi. **RCS** funziona mantenendo i set di patch (ovvero le differenze tra i file) in un formato speciale su disco, e può quindi ricreare l'aspetto di qualsiasi file in qualsiasi momento sommando tutte le patch.
Il secondo tipo di VCS (**Fig. 2**, b) è il **centralized VCS**. Il **centralized VCS** è nato dalla necessità di collaborare con sviluppatori su altri sistemi. Per gestire questo problema sono stati sviluppati i Centralized Version Control Systems (CVCS). Questi sistemi, come CVS, Subversion e Perforce, utilizzano un server centrale che contiene tutti i file sottoposti a controllo di versione e un certo numero di client che estraggono i file da questo luogo centrale. Per molti anni, questo è stato lo standard per il controllo di versione. Questa configurazione offre numerosi vantaggi rispetto ai VCS locali. Ad esempio, permette a tutti di avere una visione chiara di cosa stanno facendo gli altri membri del progetto, gli amministratori hanno un controllo dettagliato su chi può fare cosa e amministrare un CVCS è molto più semplice rispetto alla gestione di database locali su ogni client. Tuttavia questa configurazione presenta anche alcuni seri svantaggi. Il più evidente è il singolo punto di errore rappresentato dal server centralizzato: se il server si blocca per un'ora durante quel tempo nessuno può collaborare, salvare le modifiche oppure se il disco rigido che contiene il database centrale si corrompe e non sono stati eseguiti backup adeguati, si perde tutto: l'intera cronologia del progetto, ad eccezione di qualsiasi snapshot locale che le persone hanno sui loro computer. I VCS locali soffrono dello stesso problema, ovvero ogni volta che si concentra l'intera cronologia del progetto in un unico posto si rischia di perdere tutto.
Infine, l'ultimo tipo di VCS è il **Distributed Version Control System** (DVCS, **Fig. 2** c). In un DVCS, come Git, Mercurial o Darcs, i client non si limitano a controllare l'ultima istantanea dei file; piuttosto, clonano completamente il repository, inclusa la sua cronologia completa. Di conseguenza, se un server si guasta e i sistemi stavano collaborando tramite quel server, qualsiasi repository client può essere copiato nuovamente sul server per ripristinarlo, questo perché ogni clone è a tutti gli effetti un backup completo di tutti i dati. Inoltre, molti di questi sistemi gestiscono molto bene la presenza di diversi repository remoti, permettendo di collaborare con vari gruppi di persone in modi differenti all'interno dello stesso progetto. Questo consente di configurare diversi tipi di flussi di lavoro che non sono possibili nei sistemi centralizzati, come i modelli gerarchici. Per i vantaggi che offre questa ultima tecnologia è stato scelto **Git** [13], che è anche uno dei software di controllo versione più utilizzati dagli sviluppatori (Fonte: bitbucket.org).
Git [13], è un sistema di controllo di versione distribuito, veloce e scalabile, adatto a gestire varie tipologie di documenti, non solo codice sorgente. Offre un set di comandi ricco e potente, che include sia operazioni di alto livello che di basso livello per la gestione dei flussi di lavoro.
I principali vantaggi dell'utilizzo di Git includono la possibilità di lavorare in team sullo stesso progetto in modo coordinato e organizzato, consente di lavorare su diversi branch (rami) dello stesso progetto sincronizzando il codice con comandi specifici e funzioni che verranno elencati di seguito:

- **Snapshots**: Il modo in cui **Git** organizza i dati riguarda una serie di **Snapshot** di un file system in miniatura. Ogni volta che vengono salvati i dati del progetto, **Git** "scatta una foto" di come appaiono tutti i file in quel momento e memorizza un riferimento a quello **Snapshot**. Per essere efficiente, se i file non sono stati cambiati, Git non memorizza di nuovo il file, ma solo un collegamento al file identico precedente che ha già memorizzato. Per concludere si può pensare che Git gestisca i suoi dati come un flusso di **Snapshot**.

- **Stati**: I 3 stati che coinvolgono i dati gestiti da **Git** sono:

  - `modified` significa che il file è stato modificato ma non è ancora stato "committato" al database.

  - `staged` significa che si è contrassegnato un file modificato nella sua versione corrente per inserirlo nel prossimo **Snapshot** del commit.
  - `committed` significa che il dato è memorizzato con successo nel database locale.

- **Commit**: Un commit in Git rappresenta uno snapshot del progetto in un determinato momento, con i dati salvati con successo nel database locale. Quando viene effettuato un commit, Git salva uno snapshot dell'intero repository, tuttavia per ottimizzare lo spazio di archiviazione, Git memorizza solo i file che sono stati modificati (quelli marcati come `staged`) rispetto al commit precedente. Per i file non modificati, Git crea semplici riferimenti ai dati già esistenti, garantendo così un utilizzo efficiente dello spazio.

- **Repository (Repo)**: Un repository Git è dove **Git** memorizza i meta dati e il database del progetto. Questa è la parte più importante di Git, perché vengono salvati tutti i commit (con conseguenti snapshot). Una repository può essere locale (dunque sul proprio file system) o remota (su un server come **Github** [14], GitLab, Bitbucket).

- **Lavoro in locale**: La maggior parte delle operazioni svolte con **Git** avviene in locale, rendendolo estremamente distribuito e veloce. Questo perché, una volta scaricato lo stato corrente del progetto da remoto, si ha tutto il necessario per poter lavorare. Solo successivamente, quando si desidera aggiornare il repository remoto, sarà necessaria una connessione a Internet. Generalmente Git aggiunge dati allo stato corrente del progetto, mentre per poter eliminare dei dati importanti sono necessarie operazioni più laboriose, in questo modo una volta effettuato il commit dei dati diventa molto difficile perderli, specialmente se si esegue l'operazione di commit con regolarità.

- **Clone**: Quando si inizia a lavorare su un progetto gia esistente in remoto è necessario fare una operazione di **clone**. Questa operazione consiste nel creare una copia completa del repository remoto sul proprio computer locale.

- **Branch**: Un branch (ramo) è una linea di sviluppo indipendente che rende il progetto più ordinato e strutturato. Il branch principale è solitamente chiamato `main` o `master`, ma è possibile creare branch aggiuntivi per lavorare su nuove funzionalità o correzioni di bug senza interferire con il branch principale.

- **Fetch**: Il comando `git fetch` scarica le modifiche dal repository remoto senza fonderle automaticamente nel branch corrente.

- **Merge**: Il merge è l'operazione che combina le modifiche da due branch differenti. Solitamente, un branch di feature (o dev) viene fuso nel branch principale una volta completato e testato.

- **Pull**: Il comando `git pull` aggiorna il repository locale con le modifiche ottenute dal repository remoto. Combina i comandi `fetch` (scarica le modifiche) e `merge` (integra le modifiche).

- **Push**: Il comando `git push` invia i commit locali al repository remoto, aggiornando il repository remoto con le modifiche apportate localmente.

Solitamente nello sviluppo di codice si utilizza un **Branch** principale chiamato `main` e poi uno o più **Branch** per lo sviluppo `dev`, per il testing `test` o per specifiche funzionalità. La creazione di un nuovo **Branch** permette di lavorare in modo indipendente su un progetto senza intaccare il lavoro di altri colleghi. Una volta che le modifiche nel **Branch** `dev` sono state completate e testate avviene una fase di riunione con il **Branch** `main` tramite la funzione di **Merge**. Per fare invece un semplice rollback del codice basterà tornare ad un commit precedente scartando quello che dà problemi, grazie al comando `Reset`.

2.3 Github

Github [14] è una piattaforma basata su cloud che permette di archiviare, condividere e collaborare nella scrittura di codice. Memorizzare il codice in un "repository" su GitHub offre numerosi vantaggi, tra cui:

- Mostrare o condividere il lavoro svolto.
- Tenere traccia e gestire le modifiche al codice nel tempo.
- Consentire ad altri di rivedere il codice e suggerire miglioramenti.
- Collaborare a un progetto condiviso senza preoccuparsi che le modifiche abbiano un impatto sul lavoro dei collaboratori prima di essere pronte per l'integrazione.

Il lavoro collaborativo, una delle caratteristiche fondamentali di GitHub, è reso possibile dal software open source Git, su cui si basa GitHub. In sostanza, GitHub è molto più di un'interfaccia web per utilizzare Git: è una piattaforma completa che offre funzionalità avanzate per la gestione del codice, la collaborazione tra sviluppatori, la revisione del codice, l'integrazione continua, e molto altro, accessibile da qualsiasi dispositivo connesso a Internet. Per poter lavorare con Github in team è molto semplice, basta creare un account per ogni sviluppatore e creare dei progetti privati visibili solo alle persone di interesse. Da qui poi è possibile fare diverse operazioni grazie ad una interfaccia web o api (ad esempio l'integrazione che offre Visual Studio per lavorare con Github) semplice e funzionale che soddisfa le varie richieste di un programmatore.
Vediamo adesso come viene gestito con **Visual Studio** l'integrazione di codice su Github e in locale.

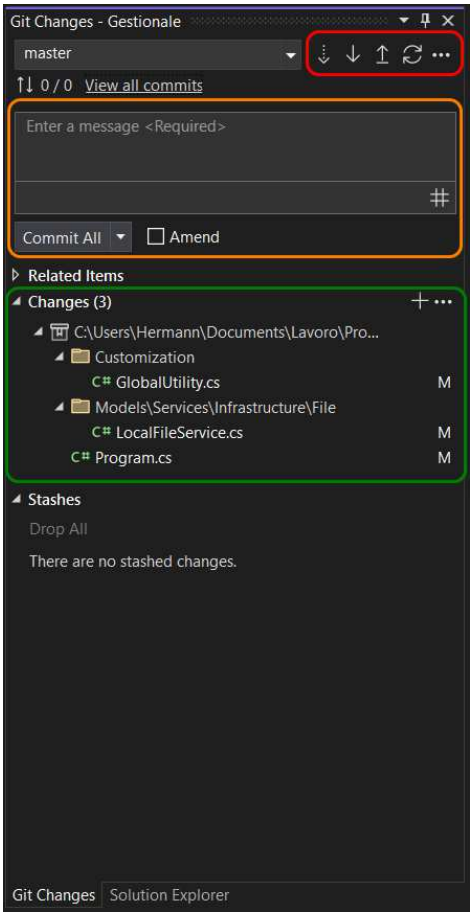


Fig. 3. Finestra di gestione Git/Github

Nella figura sono state evidenziate con dei colori le schede e le aree che verranno discusse in questo elenco:

- Directory remota: Questa sezione mette a disposizione tutte le funzionalità per poter gestire una repository remota. Partendo da sinistra con la freccia tratteggiata che punta verso il basso si effettua una operazione di `fetch`, poi la freccia a linea continua che punta verso il basso permette di fare un `pull`, operazione essenziale per poter prendere le modifiche fatte da un altro collaboratore o fatte su un altro dispositivo e integrarle subito con il progetto in locale. La freccia in linea continua che punta verso l'alto viene utilizzata per fare un `push` del branch specificato a fianco, nel branch omonimo in remoto. Poi le 2 frecce che formano un cerchio servono per poter fare un `Sync` e cioè una operazione di `pull` e `push`. Infine è presente un meno overflow a tre puntini per poter effettuare altre operazioni più specifiche.
- **View all commits**: Permette di vedere tutti i commit di un particolare branch, di fare dei `merge` tra branch in locale e molto altro.
- Commit: Sezione per poter gestire un commit, con una casella di testo per specificare il messaggio che rappresenta il commit stesso.
- Gestione file: da questa sezione è possibile gestire i file da includere nei commit. Tra le varie operazioni possibili quella più interessante è eseguire un rollback del singolo file, ripristinandolo allo stato del commit precedente.

2.4 Postman

Postman [15] è un'applicazione client per API che consente agli sviluppatori di progettare, testare, documentare e simulare le API in modo semplice ed efficiente. Questa app permette di creare richieste HTTP altamente personalizzabili, variando tra richieste GET, POST, HEAD, DELETE, e altre ancora; manipolare gli header e il body delle richieste attraverso interfacce grafiche sviluppate ad hoc e gestire il meccanismo di autenticazione previsto dallo standard adottato (E.g. ouath/2). Inoltre, è possibile analizzare le risposte ricevute in vari formati, come "Pretty" (JSON, XML, HTML, Text, Auto), "Raw", "Preview" e "Visualize". Un'altra caratteristica molto interessante è quella di poter definire una "Collection", ossia un insieme di richieste e risposte effettuate, con l'obbiettivo di produrre una documentazione. L'utilizzo di questo software è stato fondamentale per testare l'efficacia dei servizi di manipolazione dei dati, specialmente in vista dello sviluppo futuro di un'app mobile di manutenzione che si interfacci con gli stessi database della Web App.

2.5 SSMS

Un componente fondamentale per la gestione dei dati nel database è un Database Manager, ossia uno strumento software con integrato un DBMS che permette la manipolazione e visualizzazione dei dati. Questo strumento non solo consente di effettuare query e operazioni sui dati, ma offre anche funzionalità avanzate come il monitoraggio delle prestazioni, il backup e il ripristino dei database, e la gestione della sicurezza e degli accessi. È essenziale per eseguire operazioni di diagnostica e controllo sui dati nel caso in cui l'applicazione incontri errori o modifichi erroneamente i dati. La scelta di un particolare Database Manager dipende dal DBMS di riferimento, come ad esempio PostgreSQL, SQL Server, Oracle, ecc. In questo progetto, è stato utilizzato **SQL Server Management Studio** [16] (SSMS) poiché il DBMS scelto è SQL Server. SSMS fornisce una piattaforma robusta e intuitiva per amministrare e ottimizzare il database, garantendo un elevato livello di controllo e affidabilità nella gestione delle risorse. Durante lo sviluppo del programma di manutenzione, questa applicazione è stata utilizzata per monitorare lo stato del database in tempo reale mentre venivano inseriti nuovi dati dall'app. È stata inoltre impiegata la funzionalità di editing delle tabelle per modificare rapidamente i dati errati, senza la necessità di scrivere query SQL. Oltre alla modifica dei dati, sono state eliminate righe che risultavano errate o che potevano interferire con l'applicazione di migrazioni, con eventuali modifiche al design di alcune tabelle.

2.6 Docker e Trefik

Gli ultimi componenti software presentati soddisfano una delle ultime esigenze quando si sviluppa un software, ovvero la pubblicazione di un'applicazione. Per pubblicare un software, esistono diverse piattaforme che vanno dalle più datate alle più recenti, seguendo il progresso tecnologico. Innanzitutto, è importante distinguere la tipologia di server tra server locali, dunque di proprietà dell'azienda e server in cloud, gestiti da terze parti. È inoltre fondamentale differenziare tra l'utilizzo di una macchina fisica, ovvero un server con tutte le sue caratteristiche hardware, prestazionali e di efficienza legati alla natura e tipologia della macchina considerata, e l'utilizzo di una macchina virtuale, che rappresenta una "porzione" di una macchina fisica e offre un numero limitato di risorse rispetto a un server fisico completo. Questa soluzione è spesso proposta da grandi aziende come AWS, Google Cloud e Azure, che forniscono macchine virtuali anziché fisiche, operando in ambienti cloud.
L'ultima tecnologia considerata è quella della containerizzazione e dei microservizi. Questa tecnologia consente un'esecuzione più efficiente delle applicazioni sulla piattaforma server, poiché utilizza meno risorse rispetto alle tecnologie precedenti. Inoltre, i container operano tutti sullo stesso sistema operativo, offrendo vantaggi significativi in termini di risorse, velocità nello scambio di informazioni e sicurezza, visto che i container nascono completamente chiusi e per farli interagire con l'esterno si vanno a specificare le porte da aprire, similmente a dei firewall. Un software che implementa questo concetto è **Docker** [17], che, grazie ai suoi componenti che esamineremo tra poco, fornisce un servizio di deploy delle applicazioni molto più rapido rispetto alle configurazioni tradizionali. Con semplici comandi, è possibile avviare da zero sia microservizi semplici, come un gestore di password o un server di posta, sia reti di container per la gestione di web app molto più complesse. **Docker** è una piattaforma open source per lo sviluppo, la distribuzione e l'esecuzione di applicazioni. Consente di separare le applicazioni dall'infrastruttura, facilitando una distribuzione rapida del software. Con Docker si può gestire la propria infrastruttura come se fosse un'applicazione. Sfruttando le metodologie di Docker per la distribuzione, il testing e il deployment del codice, è possibile ridurre significativamente il tempo tra la scrittura del codice e la sua esecuzione in produzione.
Vediamo i componenti principali di Docker.

- **Docker Deamon**: noto anche come `dockerd`, è un componente fondamentale dell'architettura Docker. Esso gestisce tutte le attività Docker sul sistema host, inclusi il ciclo di vita dei container, la gestione delle immagini, delle reti e dei volumi. In sostanza un daemon si occupa del lavoro di creazione, esecuzione e distribuzione dei container Docker. Un daemon può anche comunicare con altri daemon per gestire i servizi Docker.

- **Docker Client**: Il client Docker (`docker`) è il principale strumento con cui gli utenti interagiscono con Docker. Quando si eseguono comandi come `docker run`, il client invia questi comandi al demone Docker (`dockerd`), che li esegue. Il comando `docker` utilizza la Docker API per comunicare con il Deamon. Inoltre, il client Docker può interagire con più di un demone, permettendo una gestione flessibile e scalabile delle risorse Docker. Un altro esempio di docker client è il **Docker Compose** che permette di lavorare con delle applicazioni che consistono in un insieme di container.

- **Docker Desktop**: il Docker Desktop è un'applicazione facilmente installabile per gli ambienti Mac, Linux e Windows. Questa app consente di creare, condividere ed eseguire applicazioni e microservizi containerizzati molto velocemente, fornendo dunque un servizio di **Docker Client** guidato e di facile utilizzo.

- **Docker Compose**: Un file "docker-compose.yml" è un file YAML utilizzato per definire e gestire applicazioni multi-container. Docker Compose permette di definire come i container dovrebbero interagire tra loro, automatizzando il deployment e la gestione di applicazioni complesse.

- **Image**: Un'immagine Docker è un pacchetto leggero e portabile che contiene tutto il necessario per eseguire un'applicazione, inclusi il codice, le librerie, le dipendenze e le configurazioni. Le immagini sono costruite seguendo un file di istruzioni chiamato Dockerfile e possono essere utilizzate per creare container.

- **Dockerfile**: Un Dockerfile è un file di testo che contiene una serie di comandi per assemblare un'immagine Docker. Definisce l'ambiente in cui verrà eseguita l'applicazione, specificando i pacchetti da installare, le configurazioni da applicare e le operazioni da eseguire. Quando viene modificato il Dockerfile e viene ricompilata l'immagine, vengono ricompilati solo i layer che sono stati modificati. Questo è uno dei motivi per cui le immagini sono così leggere, di piccole dimensioni e veloci, se confrontate con altre tecnologie di virtualizzazione.

- **Container**: Un container Docker è un'istanza in esecuzione di un'immagine Docker. I container sono isolati tra loro e dal sistema operativo host, garantendo un livello di astrazione tra i due. Ogni container viene avviato a partire da una singola immagine e può essere visto come un processo in esecuzione con tutto il necessario per eseguire l'applicazione specificata nell'immagine.

- **Registry**: Un registry è un servizio dove vengono memorizzate e distribuite le immagini Docker. Docker Hub è il registry pubblico più noto, ma è possibile anche configurare registri privati. I registri consentono di scaricare e caricare immagini Docker, facilitando la condivisione e la distribuzione del software.

- **Volume**: Un volume è un meccanismo per salvare i dati generati dai container. I volumi permettono ai dati di sopravvivere anche se i container vengono distrutti e ricreati. Sono utili per archiviare dati che devono essere conservati a lungo termine, come i database.

- **Network**: Docker permette di definire reti virtuali per i container, consentendo loro di comunicare tra di loro e con l'esterno. Docker gestisce automaticamente le configurazioni di rete, ma offre anche la possibilità di creare reti personalizzate.

Prima di spiegare come è stata strutturata la rete di container è necessario introdurre l'ultimo strumento software e cioè il reverse proxy **Traefik** [18].
Traefik [18] è un moderno reverse proxy open source progettato per facilitare la pubblicazione di applicazioni e gestire al meglio lo smistamento delle richieste provenienti dall'esterno verso una rete di uno o più server (nel nostro caso, container Docker). Per funzionare correttamente nel nostro contesto, Traefik richiede un file "traefik.yml" per la configurazione del reverse proxy e un file "docker-compose.yml" per specificare come deve comportarsi il container Traefik.
Nell'app sviluppata sono stati utilizzati tre container i cui comportamenti sono stati definiti tramite i file "docker-compose.yml". In particolare, sono stati utilizzati un container per l'applicazione web, uno per SQL Server (per i dati salvati a DB) e un terzo container per **Traefik**, utilizzato per definire la rete tra i vari container e l'esterno.
Infine assumendo di aver specificato i vari container e la network Docker, vediamo i comandi per potere fare il deploy dell'app:

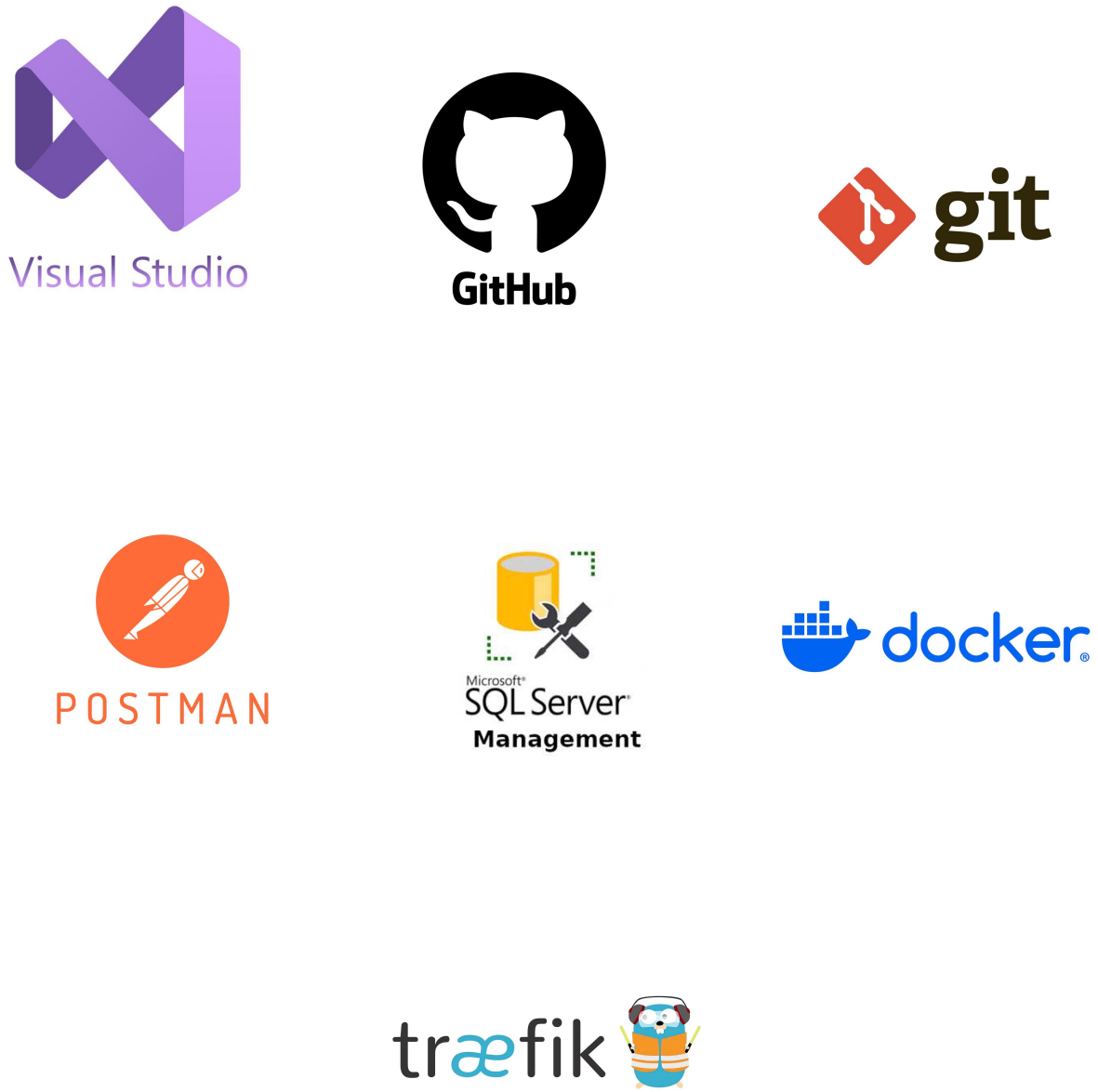
- `docker build -t name:tag .`
  Il primo comando crea un'immagine Docker a partire da un Dockerfile nella directory corrente (infatti il docker file è stato messo allo stesso livello del file ***.csproj**). L'opzione `-t` assegna un tag all'immagine, che verrà usata come riferimento nel comando successivo.

- `docker push name:tag`
  Il secondo comando carica l'immagine taggata `name:tag` nel repository Docker remoto specificato, rendendola disponibile per altri sistemi. Per riconoscere il repository remoto, è stata utilizzata in background l'applicazione **Docker Desktop**, con l'accesso effettuato all'account contenente gli hub Docker pertinenti. In questo modo, quando si esegue il comando `push`, l'app Docker Desktop individua automaticamente il repository remoto, facilitando il caricamento dell'immagine.

In conclusione si fa un accesso in ssh alla macchina remota che contiene il Docker hub, si naviga nelle cartelle che contengono i container e si eseguono gli ultimi 2 comandi:

- `sudo docker-compose pull`
  Questo comando scarica le immagini Docker specificate nel file `docker-compose.yml` dal repository remoto, questo è utile per aggiornare le immagini alla loro ultima versione disponibile nel repository prima di eseguire i container. L'opzione `sudo` viene utilizzata per eseguire il comando con privilegi di amministratore, garantendo così che tutte le operazioni necessarie possano essere completate senza restrizioni di permessi.

- `sudo docker-compose up -d`
  L'ultimo comando avvia i container definiti nel file `docker-compose.yml` in backgorund (detached mode), permettendo al terminale di essere utilizzato per altri comandi. Questo comando è utile per avviare l'intera applicazione Docker composta da più servizi/container definiti nel file di configurazione.



Loghi degli strumenti software presentati

3 Progettazione e sviluppo con ASP.NET Core

Questo capitolo tratta i costrutti fondamentali per la realizzazione di una Web App con ASP.NET Core. In ogni sezione verrà spiegato, prima di tutto, l'obiettivo concreto da raggiungere, seguito da una descrizione delle tecnologie utilizzate per raggiungere tale obiettivo. L'ordine in cui vengono presentati i vari concetti segue la sequenza di sviluppo delle diverse funzioni dell'applicazione, permettendo un migliore collegamento tra i vari argomenti e un flusso di spiegazione più coerente e vicino all'approccio pratico.

3.1 Progettazione

La progettazione è una parte essenziale per la creazione di una Web App. Definendo solide basi e idee chiare, è possibile sviluppare il software in modo più lineare ed efficiente. Vediamo quindi gli step principali per una buona progettazione del software.
La prima fase riguarda solitamente la **raccolta dei requisiti**. In questa fase si esaminano le esigenze che il programma deve soddisfare e gli obiettivi che si vuole raggiungere. Spesso questa fase avviene tramite un incontro con i clienti che hanno commissionato il lavoro o, nel caso di un prodotto proprietario, tramite un incontro aziendale. Durante questo incontro si valutano le entità principali in gioco e per ognuna di queste si fissano delle funzionalità di base, suddividendo i requisiti in:

- **Requisiti funzionali**: Compiti e operazioni concrete che l'app deve svolgere.
- **Requisiti non funzionali**: Aspetti riguardanti le prestazioni, la sicurezza, la scalabilità, ecc.

Nel caso di un programma che deve affrontare dei competitor si effettua anche un'**analisi della concorrenza**, individuandone i punti di forza e di debolezza per ottenere una differenziazione e un miglioramento. Successivamente, viene svolto uno studio sul pubblico target per comprenderne preferenze, esigenze e comportamenti.
L'ultima fase di questa parte preliminare di progettazione riguarda la **pianificazione del progetto**. In questa fase vengono stabiliti i principali milestone del progetto e le relative scadenze, viene fatta una stima delle risorse necessarie (ad esempio sviluppatori, designer, tester, ecc.) e infine viene creato un **backlog** dettagliato.
Dopo aver definito queste informazioni di base, si passa alla **definizione dell'architettura**, dove si sceglie la tecnologia da utilizzare (in questo caso ASP.NET Core), si progetta concretamente il programma con la suddivisione in moduli, si definisce la struttura del database e le API necessarie. Infine, si effettuano considerazioni sulla scalabilità per valutare la crescita dell'applicazione in funzione di un possibile aumento di utenti o di dati.
Queste prime fasi portano allo sviluppo del **back end**, fondamentale e necessario per lo sviluppo della seconda parte di un'applicazione, ossia il **front end**. In questa fase ci si occupa di definire i principali flussi dell'app per progettare l'interfaccia utente (UI) e l'esperienza utente (UX). Un altro tema molto importante è quello della sicurezza e della gestione dell'utenza. In questa fase vengono sviluppati i ruoli degli utenti (se necessarie diverse utenze) e vengono identificate potenziali minacce alla sicurezza, che si vogliono risolvere con misure adeguate.
La fase finale prevede lo svolgimento di test di usabilità (anche con utenti reali, se possibile) e verifiche del funzionamento generale per raccogliere feedback e migliorare il design.
Per la progettazione di questo programma di manutenzioni è stata organizzata una riunione aziendale, poiché si trattava di un software da utilizzare internamente per sopperire alla mancanza di alcune funzionalità descritte nel paragrafo 1.2. Nella fase iniziale di **raccolta dei requisiti** sono state utilizzate semplici lavagne o fogli. Una volta stabilite le funzionalità principali (sez. 1.2), si è passati alla stesura dei vari task (Macro funzionalità) da portare a termine utilizzando la piattaforma **Trello** [32] per la definizione dei vari **backlog**. Successivamente, si è definita l'architettura del progetto con alcuni **schemi UML** per rappresentare le prime entità da sviluppare. Questi schemi sono stati poi utilizzati come riferimento durante tutto lo sviluppo del progetto, permettendo di aggiungere o modificare nuove entità senza affidarsi unicamente al codice, ma mantenendo una visione globale dell'architettura.
Lo sviluppo delle varie funzionalità è avvenuto seguendo una modalità di incontri **agile** di base, ossia riunioni di 15-30 minuti ogni 2-3 giorni, durante le quali si faceva il punto della situazione e si analizzavano i progressi rispetto all'ultimo incontro. Questo metodo presenta numerosi vantaggi, tra cui la possibilità di avere un riscontro

costante tra chi ha commissionato il progetto e chi lo sta eseguendo. Inoltre, nel caso di fraintendimenti tra le parti, incontri così ravvicinati permettono di correggere rapidamente eventuali errori.
Per lo sviluppo delle funzionalità utente e della grafica sono stati organizzati incontri brevi e mirati a definire aspetti specifici del compito da svolgere, in modo da completare una determinata funzionalità nel minor tempo possibile. Questa metodologia di sviluppo software, adottata da tutta l'azienda, permette uno sviluppo efficace, rapido e orientato al miglioramento continuo.

3.2 Principi di base

Prima di implementare concretamente il programma di manutenzioni, è fondamentale conoscere e comprendere i costrutti essenziali di ASP.NET Core. In questa sezione, esamineremo la base di come avviene la compilazione e l'esecuzione di una app dotnet, i file di configurazione più importanti attraverso i quali è possibile eseguire diverse operazioni, come la definizione di variabili d'ambiente e la gestione delle richieste che determina l'implementazione della relativa pipeline. Inoltre, esploreremo il concetto di servizio, le sue diverse tipologie e il modo in cui questi facilitano l'uso delle classi e delle relative funzionalità in tutto il progetto.

3.2.1 Fondamenta di ASP.NET Core

1. **Compilazione ed esecuzione in dotnet**.
   Di base, un'applicazione .NET Core può essere creata usando il comando CLI `dotnet new <template>`, dove al posto di `<template>` è possibile specificare il tipo di applicazione che si vuole creare. Nel nostro caso, si utilizza `web` per creare una applicazione ASP.NET Core vuota. Una volta creata la nuova app, saranno presenti file essenziali come **.csproj** e **Program.cs**, che verranno spiegati in seguito. Per eseguire l'applicazione, basta utilizzare il comando `dotnet run` nella cartella in cui si trova il file **.csproj**. Questo comando creerà il `Processo` **`dotnet`** sul quale viene eseguita l'applicazione, il cui compito è di fondamentale importanza (i suoi componenti principali vengono discussi nel prossimo punto). Un altro comando importante offerto dalla CLI di dotnet è `dotnet build`, che serve per la compilazione dell'applicazione. Di seguito un'immagine che illustra in modo efficace il funzionamento:

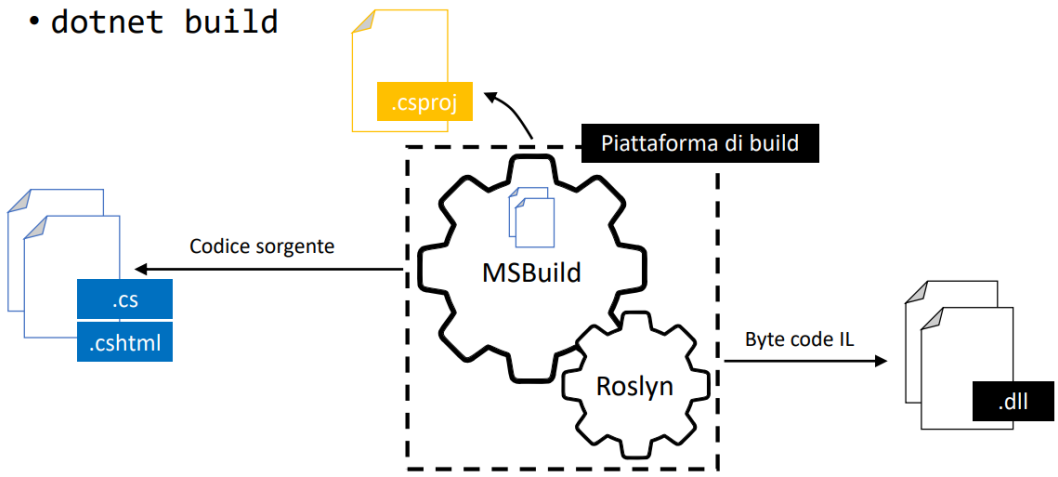


Fig. 4. Compilazione dotnet (Fonte **ASP.NET Core per tutti** [1])

Il componente essenziale è MSBuild, che organizza la build in base alle direttive scritte nel file **.csproj**. Successivamente, vengono letti i vari file sorgente, come ***.cs** e ***.cshtml** (View Razor, sezione 3.5). Una volta organizzati i file per la build, questi vengono passati a Roslyn, che è il vero compilatore della piattaforma. Roslyn produce in output file .dll scritti in un codice chiamato "Intermediate Language". Questo tipo di codice consente alle macchine di comprendere il codice sorgente scritto in un linguaggio di alto livello come il C#. Inoltre, grazie a questa funzionalità e al fatto che il codice è più vicino al linguaggio macchina, è

possibile eseguire direttamente i file .dll (passando attraverso una ulteriore fase di compilazione Just in Time grazie al componente RyuJIT, **Fig 5**) su qualsiasi macchina. In questo modo, si concretizza il concetto di portabilità e multipiattaforma, consentendo di eseguire la nostra app ASP.NET Core ovunque.

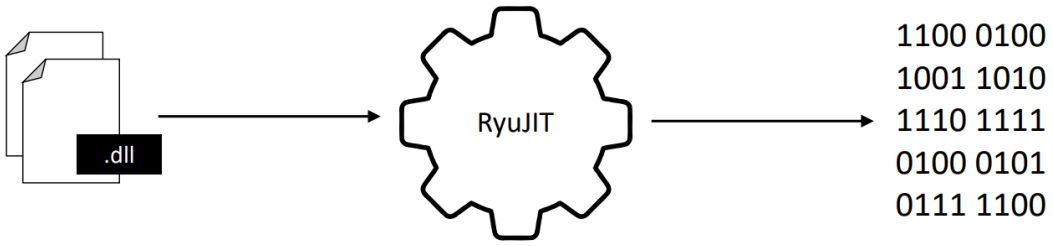


Fig. 5. Esecuzione dotnet (Fonte **ASP.NET Core per tutti** [1])

Questo procedimento di compilazione ed esecuzione avviene tutto in una volta quando si clicca il pulsante verde del "play" nella finestra a cui fa riferimento la **Fig. 1**. Altrimenti se si vuole solo compilare il progetto basterà andare nella sezione build e cliccare Build Solution.

2. **Program.cs e il web host**.
   Un altro punto chiave per lo sviluppo di un'app è comprendere le fasi di configurazione iniziali, in particolare individuare l'entrypoint del programma, cioè il punto da cui inizia la sua esecuzione. Il file di cui si parlerà è il `Program.cs`, che, come nelle più semplici applicazioni console, ospita per definizione il metodo `Main` del programma. È proprio da questo file che inizia l'esecuzione del programma ed è qui che bisogna definire gli aspetti principali su cui basare tutto il funzionamento dell' applicazione.
   La prima funzione che deve svolgere il `Program.cs` è creare un web host builder tramite l'istruzione `var builder = WebApplication.CreateBuilder(args);`. Come si può intuire (dal linguaggio C/C++), `args` è il classico array di parametri che è possibile passare al metodo `Main` durante l'esecuzione del programma tramite riga di comando. (Per approfondimenti sulla di cli dotnet consultare, **CLI** [19]).
   Ma che cos'è il web host? Per capirlo, viene proposta la seguente immagine:

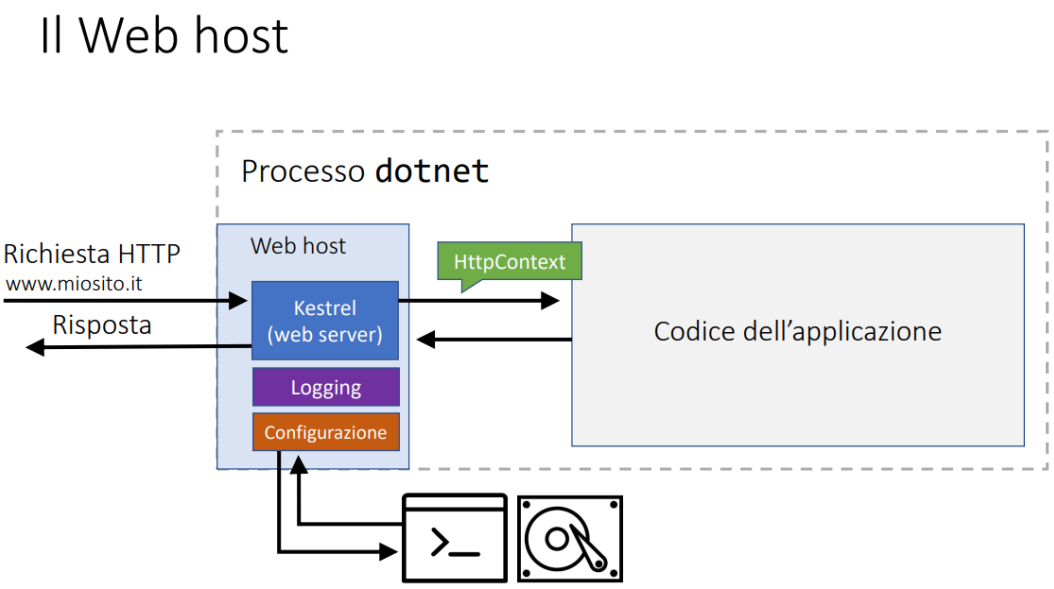


Fig. 6. Il Web Host (Fonte **ASP.NET Core per tutti** [1])

Quando un'applicazione dotnet viene eseguita tramite il comando `dotnet run`, viene avviata all'interno di un `Processo` **`dotnet`**. Questo processo deve rimanere in ascolto di eventuali richieste provenienti

dall'esterno per poterle soddisfare. Il web host è il componente che si occupa di gestire queste richieste. Infatti, è rappresentato sul margine esterno del processo dotnet nell'immagine. Il componente concreto che si occupa di soddisfare le richieste è il web server. In questo caso, si tratta di Kestrel, ma esistono anche altri tipi di web server come ad esempio HTTP.sys (che offre più funzionalità a basso livello ma che a differenza di Kestrel non è multipiattaforma) o IIS (che è il vecchio web server per applicazioni ASP.NET).
Una caratteristica molto interessante di Kestrel è che, a differenza di altri web server, "vive" nello stesso processo dell'applicazione facilitando e migliorando il modo in cui si possono gestire le richieste HTTP provenienti dall'esterno (questo aspetto raggiunge la sua concretezza con i middleware). In sostanza, un'app ASP.NET Core ha tutto il necessario all'interno di un unico processo, ovvero il `processo dotnet`. Dopo aver ricevuto una richiesta HTTP Kestrel la prende, ne fa il parsing dei dati e li de-serializza in un oggetto chiamato `HttpContext`, in modo che siano disponibili direttamente al codice dell'applicazione. In base al contenuto di `HttpContext`, verrà fornita una determinata risposta secondo le funzionalità che si intende mettere a disposizione.
Altre funzionalità utili offerte dal Web Host includono il logging, che permette di registrare messaggi di vario tipo per scopi di diagnostica o debug, e la possibilità di configurare parametri operativi da passare al Web Host per definire diverse modalità di funzionamento, per esempio tramite la riga di comando o un file di configurazione. Un'altra caratteristica importante di ASP.NET Core è la modularità. Oltre alla struttura del codice, la modularità si riflette anche nella possibilità di intercambiare i singoli componenti del web host senza intaccare gli altri meccanismi, ad esempio è possibile cambiare il tipo di web server, il servizio di logging o il modo in cui viene configurata l'applicazione.
Nelle versioni precedenti a .NET 6, il `Program.cs` era diviso in due file: uno chiamato `Program.cs` e l'altro `Startup.cs`. La differenza sostanziale tra i due era che: nel primo veniva creato e configurato il web host le cui funzionalità venivano gestite dal `Startup.cs` e all'occorrenza, venivano inseriti codici di controllo per svolgere determinate operazioni all'avvio del programma. Il secondo, lo `Startup.cs`, si occupava della definizione dei servizi, delle opzioni, della configurazione dei `DbContext` (ne è possibile avere più di uno nel caso in cui si vogliano salvare dati su più database o su database diversi con DBMS diversi) e definiva i middleware con il loro funzionamento. Questo concetto, pur anticipato, verrà trattato con spiegazioni ed esempi nella prossima sezione.

3. **File di configurazione**.
   I principali file utilizzati per la configurazione sono :

   - **launchSettings.json**: Viene utilizzato per definire impostazioni di avvio del Web Host, come ad esempio le porte su cui rimanere in ascolto o configurazioni specifiche per Docker.
   - **appsettings.json**: Contiene i parametri di configurazione principali che hanno la priorità più alta. Quando il sistema legge i dati di configurazione, consulta prima questo file.
   - **appsettings.{Environment}.json**: Definisce i dati di configurazione specifici per l'ambiente. Questo file viene preso in considerazione solo se {Environment} corrisponde all'ambiente in cui si trova l'applicazione e ha una priorità secondaria rispetto a **appsettings.json**. È possibile dichiarare diversi file per vari ambienti, come produzione, sviluppo e test, generando file come **appsettings.Production.json**, **appsettings.Development.json** e **appsettings.Staging.json**.

   Questi file sono semplici file JSON, ovvero file di testo che utilizzano la sintassi di notazione degli oggetti JavaScript per rappresentare dati strutturati. Di seguito viene fornito un esempio di **appsettings.Development.json**.

Fig. 7. appsettings.Development.json

Come si può vedere dalla **Fig. 7**, gli oggetti definiti sono di tipo chiave-valore, dove la chiave è una stringa e il valore può essere una stringa, un numero, un valore booleano, null, un oggetto JSON annidato o un array di oggetti. L'importanza di questo file risiede nella possibilità di definire costanti reperibili in tutto il progetto tramite il meccanismo di definizione di una "Option" e della **Dependency Injection** (spiegata alla sezione 3.2.3). Inoltre, è possibile sovrascrivere queste variabili (tramite il file **docker-compose.yml** visto in precedenza) quando si desidera cambiarne il valore in base all'ambiente del progetto. Per fare un esempio pratico, se durante lo sviluppo si utilizza un database di test la cui stringa di connessione è specificata nel file **appsettings.json**, è possibile sovrascrivere questo valore con la stringa di connessione del database di produzione.

4. ***.csproj**.
   Il file `NomeApp.csproj` (dove 'NomeApp' è il nome dato al progetto), è un file XML che contiene informazioni riguardo: riferimenti di librerie esterne utilizzate (ad esempio, scaricate tramite pacchetti **NuGet** [10]), come eseguire la build del progetto (ad esempio, escludere dei file dalla **compilazione**) o su come pubblicare l'applicazione in base ai framework e agli SDK utilizzati. Di seguito è mostrato un esempio:

```
<Project Sdk="Microsoft.NET.Sdk.Web">

  <PropertyGroup>
    <TargetFramework>net6.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.AspNetCore.Identity.EntityFrameworkCore" Version="6.0.16" />
    <PackageReference Include="Westwind.AspNetCore.LiveReload" Version="0.4.0" />
  </ItemGroup>

  <ItemGroup>
    <Folder Include="Areas\Admin\Services\" />
    <Folder Include="Customization\Attributes\" />
    <Folder Include="Models\Input\" />
    <Folder Include="Resources\" />
    <Folder Include="wwwroot\img\entities\22101015104019737 20e8b7b-f624-4bb0-9b91-6d60168c3800\" />
  </ItemGroup>

    <PropertyGroup>
        <GenerateDocumentationFile>true</GenerateDocumentationFile>
    </PropertyGroup>
</Project>
```

Fig. 8. *.csproj

In questo esempio, viene fornita l'SDK 'Microsoft.NET.Sdk.Web', che è quella standard per questo tipo di applicazioni. Successivamente, viene specificato il framework di destinazione, ossia il framework in cui è stata sviluppata l'applicazione (in questo caso, .NET 6). Infine, all'interno del tag 'ItemGroup > PackageReference', è possibile specificare i pacchetti NuGet utilizzati. Oltre a queste proprietà, ce ne sono molte altre che possono essere configurate per personalizzare ulteriormente il comportamento del progetto anche in base alla tipologia di ambiente (E.g. Produzione, Sviluppo, Debug, ecc).

3.2.2 Middleware

I **Middleware** [20] sono dei componenti di ASP.NET Core che sono già stati menzionati durante questa relazione, proprio quando si è parlato di come il `processo` **`dotnet`** interagisce con l'esterno. Questi componenti permettono di rendere più organizzata e modulare la modalità con cui vengono trattate le richieste, andando cosi a migliorare quella che è la funzione basilare di un Server. Procediamo subito a capire con una immagine di riferimento cosa si intende.

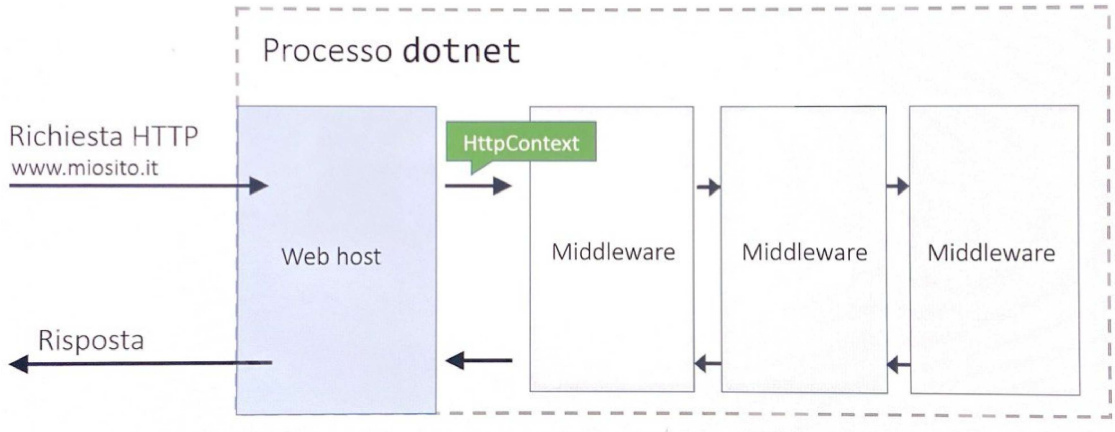


Fig. 9. I Middleware (Fonte **ASP.NET Core per tutti** [1])

La prima caratteristica che si può notare dalla **Fig 9** è che i **Middleware** sono disposti in sequenza; questo aspetto è fondamentale visto che ogni Middleware elabora la richiesta http che gli arriva dal Middleware precedente o nel caso in cui si tratti del primo Middleware, la richiesta ricevuta dal web host e la manda al middleware successivo. Nel caso in cui si tratti dell'ultimo middleware questo deve essere capace di fornire una risposta adeguata da

inoltrare al middleware precedente ripercorrendo cosi la sequenza in modo inverso fino a che non si arriva all'utente finale. Un Middleware può fare diverse operazioni come ad esempio leggere le proprietà dell'**HttpContext**, scriverci alcuni valori (ad esempio una risposta semplice nell'entity body), accedere ad un database, scrivere un file di log o collegarsi ad un web service. Esistono 3 categorie di middleware, la prima riguarda dei Middleware già presenti in ASP.NET Core e dunque sviluppati da Microsoft, la seconda riguarda dei Middleware scaricabili da pacchetti Nuget e infine la terza riguarda dei Middleware che il programmatore stesso può scrivere. Questa caratteristica rende bene il concetto di modularità visto che è il programmatore a decidere come posizionare i middleware nella sequenza e quali compiti devono svolgere. Inoltre, è evidente che non bisognerà esagerare con l'utilizzo di questi componenti perché altrimenti si rischia di rallentare il servizio di risposta rendendolo poco efficiente e molto fastidioso agli occhi dell'utente.
Nel programma di manutenzioni un esempio di utilizzo di un middleware può essere quello di gestione delle eccezioni; questo middleware non viene incluso durante lo sviluppo di codice perchè quando scatta una eccezione si vuole mantenere il programma cosi come e dunque vedere a schermo la porzione di codice che ha dato errore. Invece, in produzione non si vuole esporre il proprio codice all'utente finale e si vuole ritornare un messaggio di cortesia sullo stato dell'applicazione, di conseguenza si decide di implementare un middleware che intercetta le risposte HTTP che danno errore e che ritornerà in modo sistematico una pagina di errore predefinita con un messaggio adeguato. Questo middleware di cui si è appena discusso si chiama `UseExceptionHandler`; al suo interno sarà possibile fare un re-indirizzamento alla pagina designata e una stampa a console degli errori che sono accaduti.
Il secondo middleware di esempio ha un compito ben preciso è cioè quello di fornire all'utente dei file statici quando vengono richiesti. Il compito di questo middleware è molto semplice e cioè deve estrarre dei file da disco (Ad esempio, immagini, video, Fogli di Stile o file Javascript) e restituirli in una risposta HTTP. Tutti i file statici in questione devono essere contenuti all'interno di una cartella particolare che si chiama `wwwroot` che per definizione è la **Web Root** della applicazione; se non viene eseguita questa pratica il middleware non troverà i file e non riuscirà a ritornarli al browser richiedente. Questo middleware non solo permette di scaricare i file statici presenti all'interno di `wwwroot` ma permette anche di mappare un'altra cartella del server come contenuto di file statici in modo tale da renderli disponibili all'utente. Questa particolare funzione è stata implementata nel programma per uno scopo ben preciso e cioè quello di rendere disponibile delle immagini come file statici contenute all'interno di una cartella del server. Per poter implementare questo middleware appena discusso bisogna usare `UseStaticFiles`, con l'aggiunta di alcune configurazioni per poter svolgere la sua seconda mansione.
Infine come ultimo middleware utilizzato viene proposto il **Middleware di routing**, un componente fondamentale per far funzionare l'intero progetto. Questo middleware si occupa di mappare l'URL contenuto all'interno della richiesta HTTP nella pagina o Controller corretti. Il modo con cui il middleware di routing fa la mappatura delle pagine è assolutamente personalizzabile grazie alla creazione di pattern che rendono la struttura ad albero delle varie cartelle molto organizzata. Quest'ultima parte verrà discussa con più approfondimento nel prossimo punto.
In conclusione, se un middleware riesce a soddisfare una richiesta, interrompe la catena di comunicazione tra i vari middleware e invia immediatamente una risposta. Questo significa che eventuali middleware successivi nella catena non verranno chiamati, poiché la richiesta è stata già gestita. Da ciò si deduce l'importanza cruciale dell'ordine in cui i middleware sono disposti nella gestione di una richiesta. Ad esempio, se si desidera proteggere tutte le richieste al server con meccanismi di autenticazione e autorizzazione implementati tramite middleware, è fondamentale posizionarli prima del middleware di routing. In caso contrario, questi middleware verrebbero ignorati, rendendo inutile la loro funzione.

3.2.3 Design pattern e routing

Con i middleware discussi in precedenza, è stato menzionato solo che è possibile rispondere a una richiesta HTTP scrivendo direttamente nel corpo della risposta, il che consente di restituire semplici righe di testo. Tuttavia, l'obiettivo è ottenere una risposta composta da una pagina HTML ben formattata e per poter raggiungere questo obiettivo è stato introdotto il **middleware di routing**, un componente che mappa l'URL della richiesta a uno

specifico Controller o a una Pagina HTML. Ma cosa si intende concretamente con "Controller" o "Pagina HTML"? Approfondiamo i due principali design pattern offerti da ASP.NET Core per la realizzazione dell'interfaccia utente (UI). Il primo Desing Pattern presentato è il classico MVC (conosciuto come ASP.NET Core MVC): questo Pattern ha come scopo principale quello di dividere le responsabilità di diversi componenti software in modo tale da rendere la struttura del programma più organizzata e strutturata. Il Pattern MVC come suggerisce il nome stesso è costituito da 3 componenti:

1. **Il Model**.
   Il Model si occupa della gestione della logica applicativa e dell'accesso ai dati andando a costituire il **back end** di una applicazione. Per fare un esempio concreto, sarà proprio nel Model che si andranno a definire dei metodi per accedere al database nel caso in cui si debba interrogare o modificare dei dati, questi esempi verranno trattati (con relative porzioni di codice) nella prossima sezione.

2. **La View**.
   La View ha come scopo quello di presentare i dati in modo grafico, dunque in questo caso si tratterà di codice HTML con un misto di codice C# grazie al **View Engine Razor** [21]. Le View non devono contenere in alcun modo nessun tipo di logica applicativa (se non in casi eccezionali), ma si devono limitare a presentare i dati che gli sono stati forniti dal Controller (Per questo motivo le View vengono anche viste come "stupide").

3. **Il Controller**.
   Il Controller è il componente più importante dei 3 perché può essere visto come il **coordinatore** di tutta l'architettura; il suo compito è quello di utilizzare la logica e i dati forniti dal Model e indirizzarli alla View di interesse. Questo componente è il primo a ricevere la richiesta dell'utente e sarà lui a restituire una Pagina HTML in base alla richiesta fatta. È importante sottolineare il ruolo cruciale di questo componente, poiché è proprio quello che verrà effettivamente richiamato dal **middleware di routing** attraverso il meccanismo di routing.

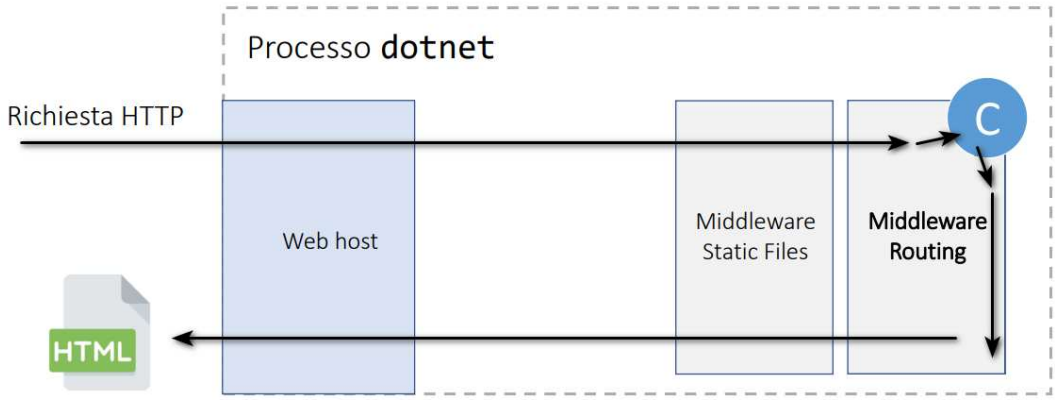


Fig. 10. Il Controller (Fonte **ASP.NET Core per tutti** [1])

Dalla **Fig. 10** si intuisce che una volta arrivata la richiesta HTTP al Middleware di routing questo andrà a cercare il Controller corretto in base alla **route** dell'URL http. Per fare un esempio i Controller non sono altro che delle semplici classi C# (disposte dentro la cartella `Controllers`), le quali derivano da una classe base `Controller` e i cui metodi vengono chiamati **action**. La mappatura dell'URL avviene attraverso il seguente pattern di `default route`: `"controller=Home/action=Index/id?"`. Questo pattern indica che nella route il primo segmento corrisponderà al nome del controller, il secondo al nome della action e il terzo ad un parametro opzionale posto direttamente in route anziché nella query per rendere l' URL più ordinato. Nel pattern sono anche presenti dei valori di default nel caso in cui non vengano specificati dei segmenti, questo permette di mappare per esempio l'indirizzo "`www.miosito.it`" in "`www.miosito.it/Home/Index`" e cioè la pagina Home del sito. Questo Desing pattern non è stato

utilizzato per il routing delle pagine ma è stato utilizzato per singole porzioni di UI chiamate **Partial View**, **View Component** e per lo sviluppo di **API**. Invece, il design pattern utilizzato per il routing delle pagine è il secondo, ovvero **Razor Pages**. Questo pattern utilizza la stessa logica presentata da MVC con una suddivisione dei componenti in 3 parti, l'unica cosa che cambia sono l'organizzazione delle cartelle che invece di essere organizzate per componenti come ASP.NET Core MVC, sono organizzate per obiettivo. Oltre a questo aspetto, anche il meccanismo di routing è un po diverso perché viene considerato il percorso delle cartelle in modo più esplicito rispetto a prima. Per poter essere raggiungibile ogni pagina Razor ha bisogno della direttiva `@Page`, dopodiché ciò che bisogna considerare è una corrispondenza tra la route della richiesta e il percorso delle cartelle. Per fare un esempio se la route di richiesta è **www.miosito.it/index**, il middleware di routing cercherà le razor page all'interno della cartella `Pages` (cartella speciale nel quale sono contenute tutte le razor pages che stanno in quel livello di directory) che sta nel livello più alto della applicazione (lo stesso del file *.csproj) e cercherà il nome della razor page che corrisponde a `index`. Come per MVC anche razor page ammette l'utilizzo di parametri in route e di organizzazioni in sotto cartelle per strutturare meglio le pagine all'interno del server.
Vediamo con una immagine la struttura delle cartelle del programma di manutenzioni.

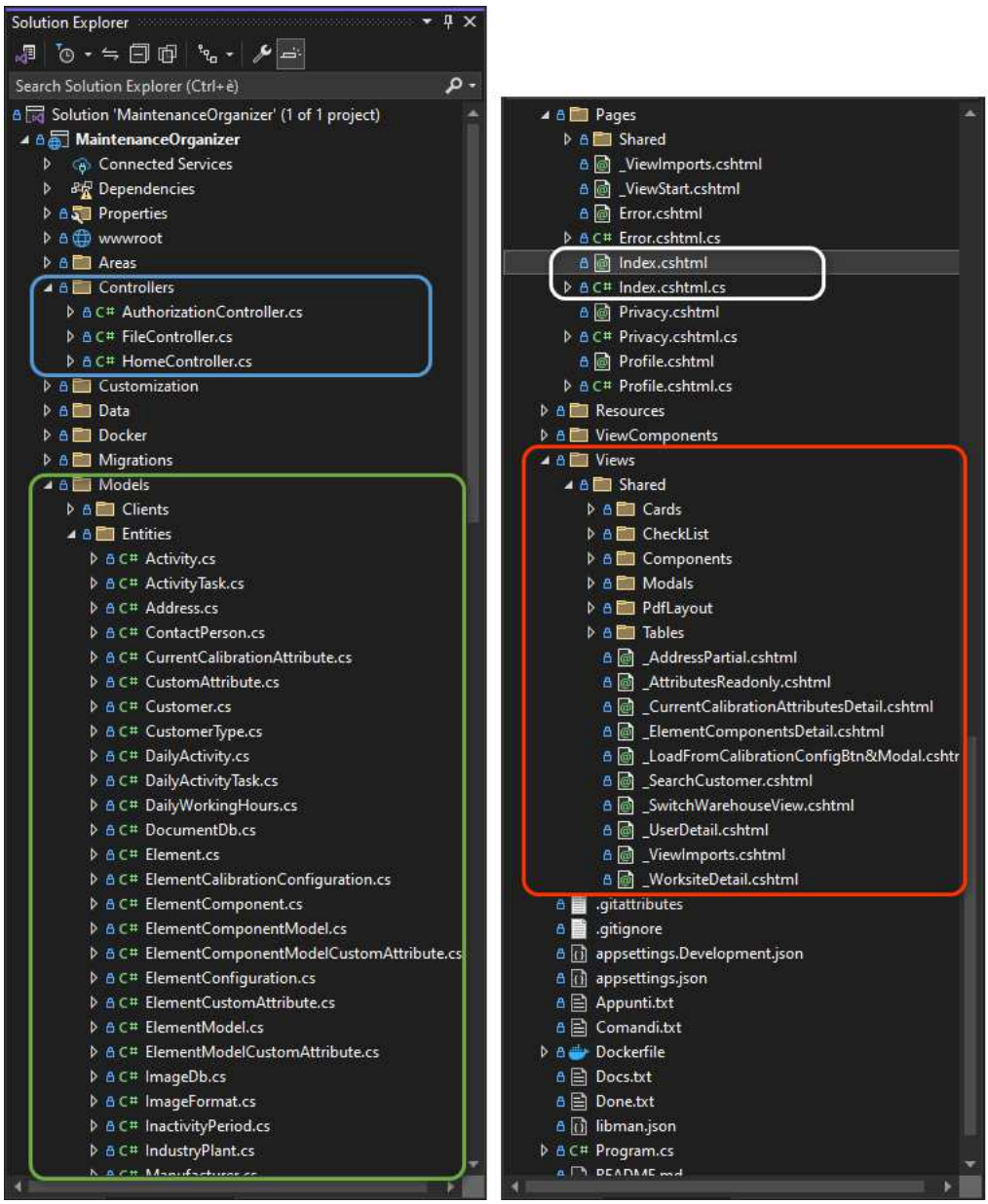


Fig. 11. Il Controller

Come si può notare per il pattern MVC le cartelle Model, Views e Controllers sono ben separate. La Cartella `Controllers` è una cartella speciale che non influisce la mappatura della route, l'importante è usare correttamente i nomi dei controller. Da notare è anche il suffisso "Controller" nel nome delle classi che non viene preso in considerazione nella mappatura dell'URL, andando cosi a mappare il file "CustomerController" semplicemente scrivendo nell'URL "Controller". Per poter determinare la `View` corretta da ritornare i Controller possono specificare il percorso della view oppure seguire anche in questo caso un pattern di cartelle. I metodi utilizzati per ritornare una UI sono `View`, `PartialView` o `ViewComponent` a seconda se si vuole ritornare una pagina intera oppure una porzione di UI da integrare in un'altra pagina. Per quanto riguarda invece Razor Pages, bisogna fare riferimento attentamente al percorso delle cartelle per arrivare alla pagina corretta, come spiegato in precedenza (Il riquadro bianco indica proprio la cartella dell'esempio fatto). Come si può intuire dall'immagine i file con estensione *.cs costituiscono il Model, i file *.cshtml sono le View e infine per MVC i Controller sono ancora delle classi in *.cs mentre per Razor Page c'è una estensione di tipo *.cshtml.cs .

3.2.4 Dependency Injection e Servizi

L'ultimo costrutto di cui si andrà a discutere per concludere questo capitolo di concetti di base è quello della Dependency Injection. La **Dependency Injection** (DI) è un design pattern utilizzato nella programmazione per gestire le dipendenze tra le classi in modo più flessibile e modulare. Invece di creare direttamente le istanze delle dipendenze all'interno di una classe, le dipendenze vengono fornite (iniettate) dall'esterno, solitamente attraverso il costruttore della classe, un metodo o una proprietà. Questo approccio consente di separare le responsabilità, facilitare il testing unitario e rendere il codice più facile da mantenere e scalare, poiché le dipendenze possono essere sostituite o configurate senza modificare la classe che le utilizza. Il **Servizio** invece, è un componente che esegue una specifica operazione o fornisce una funzionalità utilizzabile in altre parti dell'applicazione (In questo caso si tratterà quasi sempre di Classi C#). Ogni servizio può essere iniettato in un componente seguendo un preciso ciclo di vita, che verrà descritto di seguito:

- **Transient**
  In ASP.NET Core, un servizio registrato come Transient viene istanziato ogni volta che un componente ne ha bisogno e l'istanza viene distrutta subito dopo l'uso. Questo implica che, all'interno della pipeline di middleware vista in precedenza, ogni volta che un middleware o un Controller richiede un servizio Transient, ASP.NET Core creerà una nuova istanza. Questa tipologia di servizio è consigliata quando i costruttori sono leggeri e possono fornire rapidamente un'istanza al componente richiedente, minimizzando così l'impatto sulle prestazioni.

- **Scoped**
  ASP.NET Core crea una nuova istanza solo quando c'è una nuova richiesta HTTP **Fig. 10, freccia rivolta verso l'interno**) e utilizzerà questa stessa istanza all'interno di tutta la pipeline; quando la risposta HTTP (**Fig. 10, freccia rivolta verso l'esterno**) viene ritornata l'istanza del servizio viene distrutta. Questa tipologia di servizio viene utilizzata nel caso contrario del Transient, ovvero quando il costruttore impiega molto tempo per mettere a disposizione una istanza del servizio al componente richiedente. Un esempio di ciò è la registrazione del **DbContext** che verrà trattato nella sezione di Manipolazione dei dati. Un altra tipologia di servizio che potrebbe venir registrato come Scoped è un servizio che si occupa di scambiare dati tra i componenti all'interno della stessa richiesta, per esempio per tracciare dati rilevanti dell'utente: in questo esempio è necessario usare Scoped anziché Transient perché si vuole mantenere uno stato consistente del servizio all'interno di tutta la pipeline, se si usasse Transient si andrebbe a creare una istanza nuova ogni volta che viene richiesto l'uso del servizio andando cosi a perderne lo stato.

- **Singleton**
  In ASP.NET Core, un servizio registrato come singleton viene creato una sola volta e la stessa istanza viene iniettata in tutti i componenti che ne hanno bisogno, anche per richieste HTTP diverse e concorrenti. Un

esempio efficace di servizio registrato come singleton è un servizio di invio email, che è stato utilizzato nella applicazione di manutenzioni per inviare le mail di conferma registrazione, invio credenziali e altro. Il punto chiave è garantire che il server SMTP non invii più email contemporaneamente, perché se il servizio email fosse registrato come Transient o Scoped due utenti diversi che generano una richiesta di invio email potrebbero causare l'invio simultaneo di più email, poiché avrebbero istanze separate del servizio. Registrando invece il servizio mail come Singleton, è possibile implementare una coda che gestisca tutte le richieste di invio email, garantendo un controllo efficace sulla concorrenza. Questo scenario evidenzia l'importanza di assicurare che il codice di un servizio Singleton sia thread-safe.

La decisione del ciclo di vità dei servizi avviene nel file `Program.cs` (nelle versioni precedenti avveniva nello `Startup.cs`) tramite l'istruzione `builder.Services.AddModalità<{NomeServizio}>()`. Qui, `Modalità` rappresenta una delle tre opzioni disponibili, scelte in base al ciclo di vita desiderato per il servizio. Dopo che l'istanza viene distrutta, secondo le politiche appena presentate, chi si occupa di rimuovere le istanze dei Servizi dalla memoria è il **Garbage Collector**.
Infine una ulteriore funzione resa disponibile dalla **Dependency Injection** di ASP.NET Core è quella di poter rendere 2 elementi (servizio e utilizzatore) *debolmente accoppiati*. Questo significa associare ad un servizio una Interfaccia e poi andare ad utilizzare questa interfaccia nei vari Controller, classi o Pagine Razor. L'utilizzo di una interfaccia anziché di un Servizio permette di diminuire la dipendenza tra i 2 componenti in modo molto drastico. Per capire questo concetto mi servo di un esempio, implementato anche nel programma di manutenzioni. Supponiamo di aver creato un servizio che si occupa di salvare immagini in una cartella locale del server, associandolo a un'interfaccia che espone i metodi necessari per questa operazione. Dopo aver implementato i metodi richiesti, registriamo il servizio nel file **Program.cs** utilizzando l'istruzione `builder.Services.AddModalità <NomeInterfaccia,NomeServizio>()`. Successivamente, nei costruttori dei vari componenti, non utilizzeremo più il nome del servizio, ma la sua interfaccia.
La vera forza di questo approccio si manifesta quando, in futuro, decideremo di cambiare la modalità di salvataggio delle immagini, ad esempio passando da una cartella locale a un servizio AWS. Per fare ciò, basterà creare un nuovo servizio che implementa la stessa interfaccia, ma con procedure di salvataggio completamente diverse. Nel file **Program.cs** sostituiremo semplicemente l'istruzione originale con `builder.Services.AddModalità<NomeInterfaccia,NomeNuovoServizio>()`, e tutto il progetto continuerà a funzionare come prima, poiché i componenti utilizzano l'interfaccia, che è rimasta invariata.

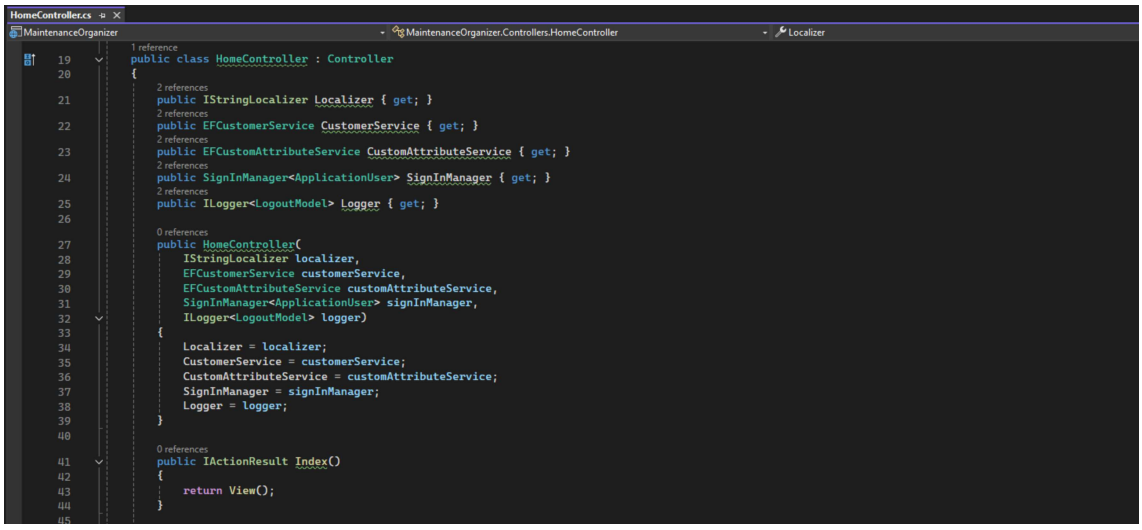


Fig. 12. Home Controller con Interfacce e Servizi iniettati

Questo argomento è fondamentale per comprendere come rendere disponibili, in qualsiasi punto del progetto, le funzionalità definite in classi create dal programmatore. La Dependency Injection ha lo scopo di rendere queste

classi accessibili nei costruttori dei Controller e nei file *.cshtml.cs delle Razor Pages. Il concetto astratto che questa tecnologia concretizza è la modularità di un progetto ASP.NET Core, infatti, grazie a questo approccio, è possibile definire da una parte una classe che gestisce la logica legata ai dati, che possono risiedere in un database o vivere per tutto il ciclo di vita del server (come nel caso delle code per il servizio mail), e dall'altra parte, definire pagine Razor o View che si limitano a visualizzare i dati senza preoccuparsi di come questi vengano gestiti.

3.3 Progettazione delle entità

Le fondamenta di qualsiasi programma software sono i dati, pertanto procedo a descrivere come vengono gestiti in questa Web App. Per semplicità, considererò solo un sottoinsieme dei dati manipolati dall'applicazione, quali i clienti (entità **Customer**), le relative sedi (entità **Worksite**), gli impianti industriali (entità **IndustryPlant**), i macchinari (o apparati industriali, entità **Element**), l'azienda che utilizza l'applicazione (entità **OwnerSettings**) e gli indirizzi (entità **Address**). Per prima cosa si andrà a considerare uno schema UML sviluppato con la Web app di Google per il disegno di diagrammi (https://app.diagrams.net/).

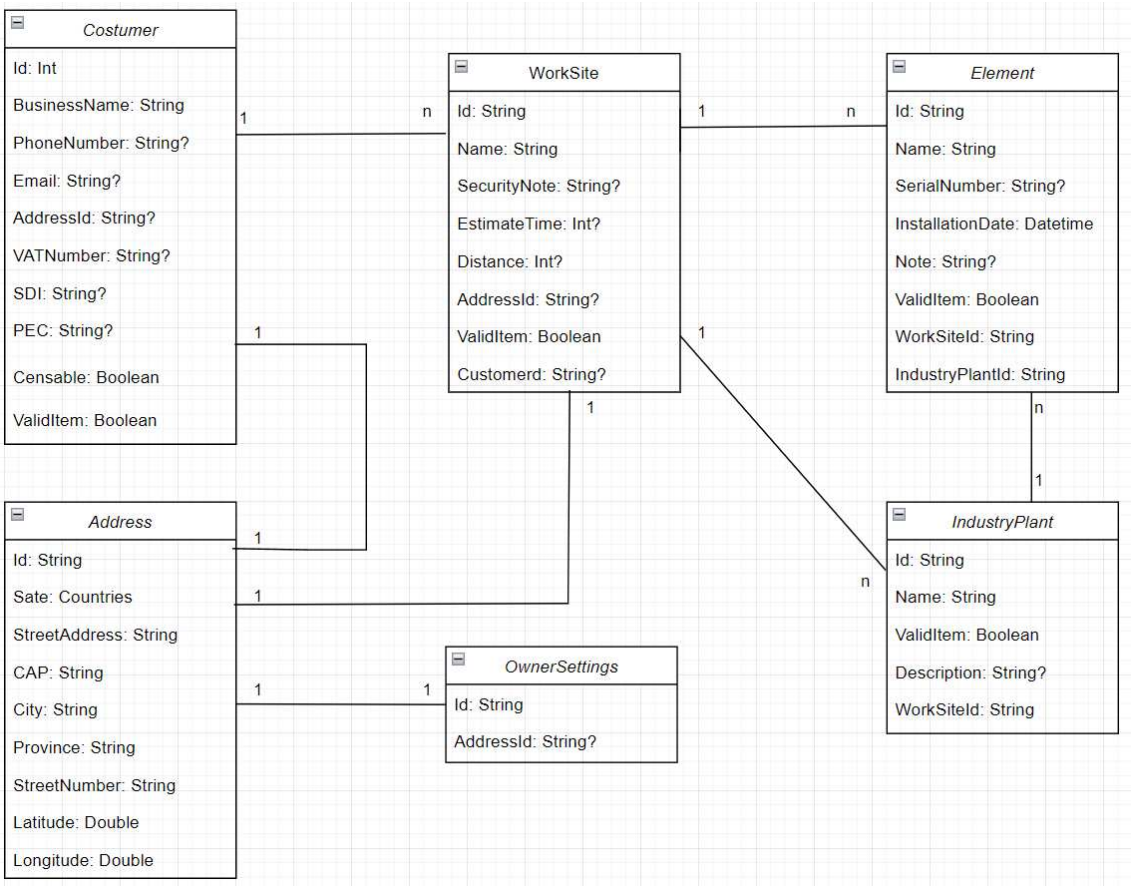


Fig. 13. Schema UML

In questo programma di manutenzioni la prima esigenza da soddifare era quella di creare una anagrafica appropriata per la gestione della clientela e dei relativi macchinari (sono stati omessi alcuni particolari e sotto entità creati dopo le 225 ore previste dall'attività lavorativa). In questo schema vengono rappresentate 5 entità tutte relazionate tra di loro; vediamo brevemente che cosa rappresentano e qual'è la loro funzione:

- **Customer**.
  Questa entità ha come scopo quella di rappresentare i clienti dell'azienda che utilizza il programma di manutenzioni; come campi sono presenti i principali dati che identificano una qualsiasi azienda, come la ragione sociale (`BusinessName`), la partita IVA (`VAT Number`), il SDI e la PEC. Il campo `Censable` è stato messo per capire se è stato fatto il censimento di quel determinato cliente (in questo caso con censimen-

to si intende un sopralluogo della struttura). Infine **Customer** ha una relazione 1 ad 1 con un indirizzo che corrisponde all'indirizzo di fatturazione del cliente e una relazione 1 a N con le sue possibili Sedi (**Worksite**).

- **Worksite**.
  Questa entità ha come scopo quello di rappresentare le sedi geografiche degli impianti di ogni cliente, un cliente deve avere almeno 1 sede. Come è intuibile, anche per questa entità è necessario un indirizzo che è stato largamente utilizzato per la gestione di una mappa della sede. Infatti sono stati messi a database anche i campi `Distance` e `EsitimateTime` che fanno riferimento alla distanza e al tempo previsto dalla sede dell'azienda che utilizza il software (`OnwerSettings`) alla sede del cliente che si sta considerando. Oltre all'indirizzo, una sede presenta una relazione 1 a N con degli impianti industriali (entità **Industry Plant**) e un'altra relazione 1 a N con dei macchinari (entità **Element**) che possono essere senza impianto industriale.

- **IndustryPlant**.
  Questa entità ha lo scopo di raggruppare diversi macchinari sotto un unico impianto e ogni sede può avere più di un impianto. Questa rappresentazione è stata fatta per poter rendere l'anagrafica più vicina alla realtà che rappresenta le sedi. Infine **IndustryPlant** ha una relazione 1 a N con i macchinari che contiene.

- **Element**.
  Questa entità rappresenta concretamente l'unità base che questo programma deve gestire e cioè gli apparati industriali. Per avere un riferimento concreto al macchinario reale sono stati inseriti il campo Nome (`Name`), il numero seriale (`SerialNumber`), la data di installazione (`InstallationDate`) e alcune note generiche (`Note`).

- **OwnerSettings**.
  Questa entità ha lo scopo di rappresentare i dati dell'azienda che sta utilizzando il software, con la possibilità (in futuro) di aggiungere nuovi dati in merito. Per il momento l'unico riferimento necessario è l'indirizzo.

- **Address**.
  L'ultima entità presentata è l'indirizzo, per cui si è deciso di creare una tabella separata al fine di garantire una migliore organizzazione delle varie entità e ottimizzare l'uso dei campi. Inserire i campi dell'indirizzo direttamente nelle tabelle di clienti, sedi e OwnerSettings avrebbe comportato il rischio di duplicazione dei campi e di avere colonne inutilmente inizializzate a null per le entità senza indirizzo. È importante notare che l'inserimento di un indirizzo non è obbligatorio, come indicato dal simbolo "?" in `AddressId: String?` nello schema UML. La tabella degli indirizzi include i campi classici per un indirizzo italiano, con l'aggiunta di latitudine e longitudine per fornire un riferimento più preciso della posizione e facilitare l'integrazione con le mappe.

Ogni entità tranne l' **Address** ha un campo `ValidItem` che sta ad indicare se l'istanza a database è consistente oppure no. Questo campo è stato scelto per evitare di eliminare definitivamente una entità dal database, preservandola nel caso sia utile in futuro. Prima di entrare nello specifico di come queste entità sono state implementate nel codice, verranno spiegati i concetti fondamentali del framework che è stato utilizzato per la manipolazione dei dati, ovvero **Entity Framework Core**. Si assume che i concetti di base dei database relazionali siano già noti.

3.4 Entity Framework Core

Entity Framework Core (EF Core) è un **Object-Relational Mapper (ORM)** leggero, estensibile e cross-platform per .NET. Consente agli sviluppatori di interagire con un database utilizzando oggetti .NET, eliminando la necessità di scrivere codice SQL manualmente. EF Core supporta operazioni CRUD (Create, Read, Update, Delete), gestione delle migrazioni del database e querying avanzato tramite **LINQ (Language Integrated Query)** [22]. Con la sua integrazione con ASP.NET Core, EF Core facilita lo sviluppo di applicazioni web moderne, garantendo efficienza e scalabilità. Di seguito verranno trattati i vari costrutti fondamentali con le loro funzioni all'interno del progetto.

L'utilizzo di una tecnologia di mapping come Entity Framework Core presenta sia vantaggi che svantaggi. Innanzitutto, come è intuibile, le operazioni di mapping necessarie per tradurre una query fortemente tipizzata con LINQ e mappare i risultati in oggetti C# comportano un aumento dei tempi computazionali rispetto all'uso diretto di query SQL tramite ADO.NET. Pertanto, è necessario valutare se la comodità offerta da EF Core giustifica l'overhead prestazionale che introduce.

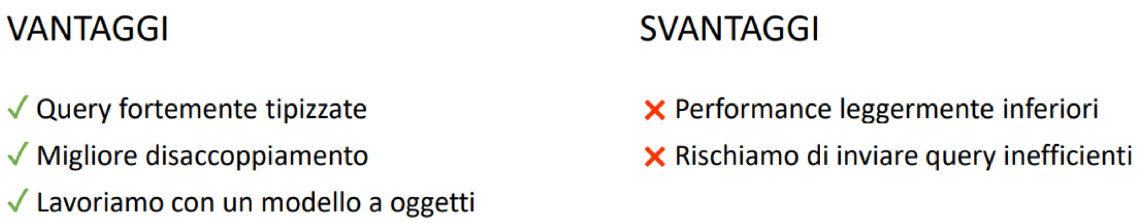


Fig. 14. Vantaggi e svantaggi di usare EFCore (Fonte **ASP.NET Core per tutti** [1])

Nel caso del programma di manutenzioni, si è scelto di utilizzare EF Core poiché le tempistiche di salvataggio dei dati non rappresentavano una priorità e va comunque sottolineato che, sebbene la differenza in termini di prestazioni sia di pochi millisecondi, questa potrebbe diventare rilevante in applicazioni in cui il real-time è un requisito cruciale. Prima di esaminare i componenti fondamentali di Entity Framework Core, è importante sottolineare che il framework offre tre modalità operative. La prima consente di eseguire un reverse engineering a partire da un database esistente, generando automaticamente le classi di interesse. La seconda modalità, invece, prevede la scrittura delle classi C# prima e, successivamente, la creazione delle tabelle del database basate su queste classi. La terza è la modalità meno usata e consiste nello scrivere manualmente il codice C# da un database già esistente, senza usare nessun comando di reverse engineering o di migrazioni. Nel caso del programma di manutenzioni, è stata scelta la seconda opzione, poiché si trattava di un progetto completamente nuovo, senza alcun riferimento preesistente.

3.4.1 I Componenti principali

I componenti principali utilizzati da Entity Framework Core sono il **DbContext**, i **DbSet** e le **classi di entità**. Ciascuno di questi elementi svolge un ruolo specifico e fondamentale per il corretto funzionamento della tecnologia. Di seguito, esamineremo il compito e l'implementazione di ciascuno di essi.
La modellazione in EF Core avviene, per prima cosa, definendo un contesto dell'oggetto che rappresenta una sessione con il database, chiamato **DbContext** e delle **classi di entità** con cui definiamo la mappatura tra classe C# e la tabella del database. Il **DbContext** è una classe che gestisce le entità e le loro relazioni, ed è responsabile della configurazione e dell'interazione con il database. Per rendere disponibile il Context in tutto il progetto bisogna usufruire della **Dependency Injection** scrivendo le seguenti righe di codice nel **Program.cs**.

```
var connectionString = builder.Configuration.GetConnectionString("DefaultConnection");

builder.Services.AddDbContext<MaintenanceOrganizerContext>(optionsBuilder =>
{
    optionsBuilder.UseSqlServer(connectionString, options => options.EnableRetryOnFailure());
});
```

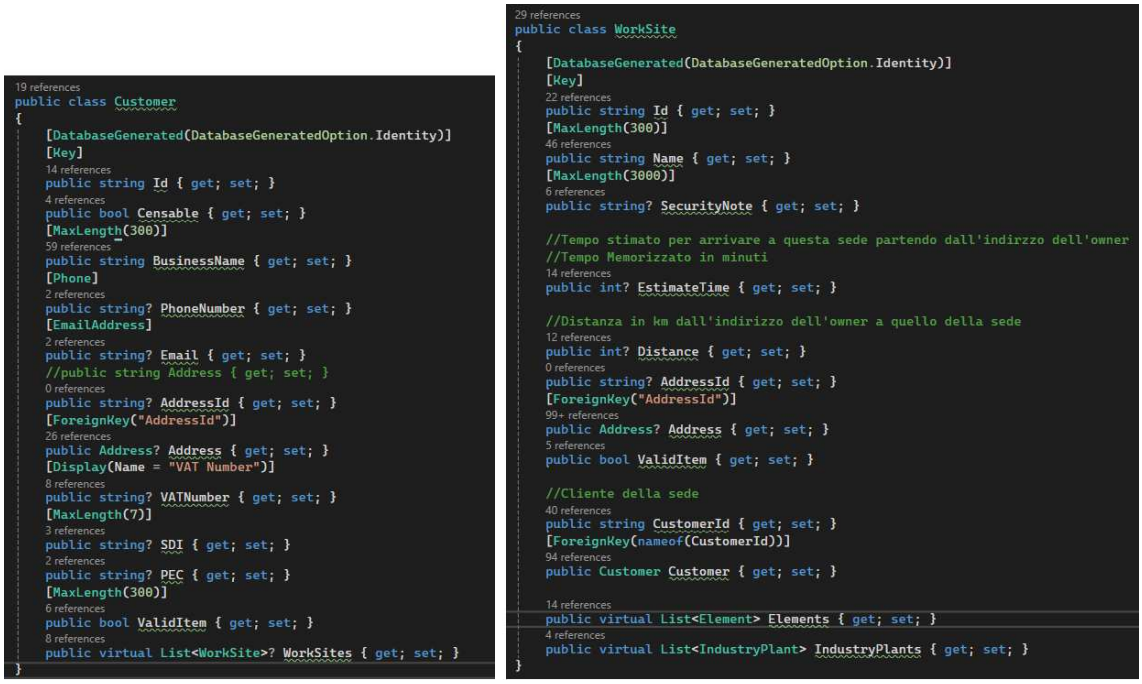
Fig. 15. Configurazione DbContext

Abbiamo così registrato il servizio `MaintenanceOrganizerContext`, il cui ciclo di vita predefinito è di tipo **Scoped**. Questo è stato scelto per garantire la creazione di una sola istanza del DbContext per ogni richiesta HTTP (le motivazioni di questa scelta sono state discusse in precedenza quando si è parlato delle tipologie di servizi). Come si può notare dalla **Fig. 15**, si è specificato il tipo di provider database utilizzato: il database a cui fa riferimento la stringa di connessione utilizza SQL Server. La scelta del provider database è assolutamente personalizzabile in base alla situazione. Dopo aver registrato il servizio che implementa il DbContext e aver definito il suo ciclo di vita, possiamo procedere alla concreta definizione di questa classe (**Fig. 16**).
Il **DbContext** deriva dal tipo `IdentityDbContext<ApplicationUser>` per avere un context che possa gestire anche le tabelle messe a disposizione da **Identity** per la gestione dell'utenza e dei relativi permessi (vedi sezione 3.6). In questa classe, per prima cosa, vengono definiti dei **DbSet<T>** che sono dei riferimenti alle tabelle del database: quando si vogliono effettuare delle operazioni su tali tabelle, basterà accedere all'oggetto dbContext e navigare verso queste tabelle. Nella prossima sezione verrà fornito un esempio di query in cui viene fatta questa operazione. Dopo aver definito il riferimento alle varie tabelle si definiscono le relazioni che ci sono tra le varie entità, all'interno del metodo `OnModelCreating`.
Questo metodo definisce le relazioni tra entità utilizzando la notazione specifica di EF Core (**API Fluent** [23]). Anche se non si è molto familiari con questa notazione, la sua sintassi, simile alla scrittura degli oggetti in C#, la rende facilmente leggibile. Ad esempio, nella relazione uno-a-molti tra **Customers** e **WorkSite**, si utilizza un oggetto `entity` (un'istanza di `EntityTypeBuilder` usata all'interno della lambda expression per costruire il mapping delle entità). Con l'oggetto `entity`, si specifica che un **Customer** ha molte sedi (**WorkSite**) tramite il metodo `HasMany`, e che ogni **WorkSite** ha come riferimento un singolo **Customer**. Successivamente, si definisce il comportamento del DBMS in caso di eliminazione di un **Customer** e cosa fare con le entità figlie **WorkSite**. In questo esempio, si è deciso di non eseguire alcuna operazione automatica: il programmatore dovrà gestire manualmente la cancellazione per evitare eccezioni, poiché eliminare un **Customer** senza rimuovere le relative sedi lascerebbe inconsistente la chiave esterna in **WorkSite**.
Questa classe è particolarmente utile per centralizzare la gestione delle entità e delle loro relazioni all'interno del progetto, garantendo una scalabilità efficiente. In questo modo, è possibile aggiungere o modificare lo schema concettuale in futuro. Oltre a definire le relazioni, nel `DbContext` è anche possibile configurare come le entità, gli enumeratori o altre particolari configurazioni di oggetti vengano mappati al database.

```
99+ references
public class MaintenanceOrganizerContext : IdentityDbContext<ApplicationUser>
{
    0 references
    public MaintenanceOrganizerContext(DbContextOptions<MaintenanceOrganizerContext> options)
        : base(options)
    {
    }

    17 references
    public virtual DbSet<Customer> Customers { get; set; }
    19 references
    public virtual DbSet<WorkSite> WorkSites { get; set; }
    8 references
    public virtual DbSet<CustomerType> CustomerTypes { get; set; }
    27 references
    public virtual DbSet<Element> Elements { get; set; }
    11 references
    public virtual DbSet<IndustryPlant> IndustryPlants { get; set; }
    9 references
    public virtual DbSet<OwnerSettings> OwnerSettings { get; set; }




    0 references
    protected override void OnModelCreating(ModelBuilder builder)
    {
        base.OnModelCreating(builder);

        builder.Entity<Customer>(entity =>
        {
            entity
            .HasMany(p => p.WorkSites)
            .WithOne(i => i.Customer)
            .OnDelete(DeleteBehavior.NoAction);

            entity
            .HasOne(p => p.Address)
            .WithOne()
            .OnDelete(DeleteBehavior.NoAction);

            entity
                .HasIndex(c => c.BusinessName)
                .IsUnique();

        });

        //Entita singola sede
        builder.Entity<WorkSite>(entity =>
        {
            entity
            .HasMany(w => w.Elements)
            .WithOne(e => e.WorkSite)
            .OnDelete(DeleteBehavior.NoAction);

            entity
            .HasOne(w => w.Address)
            .WithOne()
            .OnDelete(DeleteBehavior.NoAction);

            entity
            .HasMany(w => w.IndustryPlants)
            .WithOne(p => p.WorkSite)
            .OnDelete(DeleteBehavior.Cascade);
        });

        builder.Entity<IndustryPlant>(entity =>
        {
            entity
                .HasMany(i => i.Elements)
                .WithOne(e => e.IndustryPlant)
                .OnDelete(DeleteBehavior.NoAction);
        });



        builder.Entity<OwnerSettings>(entity =>
        {
            entity
                .HasOne(o => o.Address)
                .WithOne()
                .OnDelete(DeleteBehavior.NoAction);

            entity
               .HasOne(a => a.LogoImage)
               .WithOne(a => a.Owner)
               .IsRequired(false);
        });

    }
}
```

Fig. 16. Esempio DbContext

Per comprendere meglio queste tipologie di relazioni, è fondamentale introdurre le **classi di entità**, ovvero le classi C# utilizzate per mappare le tabelle del database sugli oggetti C#. Queste classi rappresentano i tipi specificati nel **DbSet<T>** all'interno del **DbContext**. Di seguito, viene mostrata l'implementazione delle classi **Customer** e **Worksite**; le altre entità sono state implementate seguendo i medesimi principi utilizzati per queste due classi.



```
19 references
public class Customer
{
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    [Key]
    14 references
    public string Id { get; set; }
    4 references
    public bool Censable { get; set; }
    [MaxLength(300)]
    59 references
    public string BusinessName { get; set; }
    [Phone]
    2 references
    public string? PhoneNumber { get; set; }
    [EmailAddress]
    2 references
    public string? Email { get; set; }
    //public string Address { get; set; }
    0 references
    public string? AddressId { get; set; }
    [ForeignKey("AddressId")]
    26 references
    public Address? Address { get; set; }
    [Display(Name = "VAT Number")]
    8 references
    public string? VATNumber { get; set; }
    [MaxLength(7)]
    3 references
    public string? SDI { get; set; }
    2 references
    public string? PEC { get; set; }
    [MaxLength(300)]
    6 references
    public bool ValidItem { get; set; }
    8 references
    public virtual List<WorkSite>? WorkSites { get; set; }
}
```

```
29 references
public class WorkSite
{
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    [Key]
    22 references
    public string Id { get; set; }
    [MaxLength(300)]
    46 references
    public string Name { get; set; }
    [MaxLength(3000)]
    6 references
    public string? SecurityNote { get; set; }

    //Tempo stimato per arrivare a questa sede partendo dall'indirizzo dell'owner
    //Tempo Memorizzato in minuti
    14 references
    public int? EstimateTime { get; set; }

    //Distanza in km dall'indirizzo dell'owner a quello della sede
    12 references
    public int? Distance { get; set; }
    0 references
    public string? AddressId { get; set; }
    [ForeignKey("AddressId")]
    99+ references
    public Address? Address { get; set; }
    5 references
    public bool ValidItem { get; set; }

    //Cliente della sede
    40 references
    public string CustomerId { get; set; }
    [ForeignKey(nameof(CustomerId))]
    94 references
    public Customer Customer { get; set; }

    14 references
    public virtual List<Element> Elements { get; set; }
    4 references
    public virtual List<IndustryPlant> IndustryPlants { get; set; }
}
```

Fig. 17. Esempio DbContext

Innanzitutto, queste classi sono state definite utilizzando proprietà C# che rendono gli attributi della classe più accessibili alle **API Fluent** [23] e facilitano un miglior **Data Binding**. Inoltre, molti attributi sono decorati con parametri racchiusi tra parentesi quadre, noti come **Data Annotations**. Queste annotazioni forniscono al **DbContext** indicazioni su eventuali vincoli, sia all'interno della stessa tabella (ad esempio, lunghezza massima di un campo, formattazione particolare o presenza di chiavi), sia tra più tabelle, come nel caso dei vincoli di chiave esterna.
Ad esempio, il campo `Id` è stato definito come chiave primaria semplicemente aggiungendo la Data Annotation `[Key]`. Quando questa annotazione viene utilizzata una sola volta all'interno di una classe, l'attributo viene automaticamente considerato come chiave primaria. Inoltre, è stato specificato che il campo è auto-generato grazie alla Data Annotation `[DatabaseGenerated(DatabaseGeneratedOption.Identity)]`. Sono stati definiti altri semplici vincoli **intrarelazionali**, che permettono di effettuare controlli sia di ottimizzazione dello spazio che di validazione dei campi stessi, generando un errore nel caso in cui tali vincoli non vengano rispettati.
Infine, con le Data Annotations sono stati definiti anche i vincoli **interrelazionali** per stabilire le relazioni tra le entità, relazioni che sono poi configurate completamente grazie alla **Fluent API** del DbContext. Considerando l'entità **Customer**, è stata aggiunta una lista di **Worksite** per indicare che ad ogni oggetto della classe **Customer** sono associate n sedi. Nell'entità **WorkSite**, invece, è stato definito il campo **CustomerId** (Chiave esterna) e un oggetto **Customer**, per consentire di accedere all'oggetto **Customer** partendo da un'istanza di **WorkSite**. Per definire il vincolo di chiave esterna in **WorkSite**, è stata utilizzata la Data Annotation `[ForeignKey(nameof(CustomerId))]` con riferimento all'oggetto **Customer**. Anche la classe **WorkSite** presenta relazioni 1-N, che sono state definite utilizzando lo stesso meccanismo appena spiegato (come dimostrano le due liste riferite a **Elements** e **IndustryPlants**). Tutti questi approcci e implementazioni possono essere estesi a qualsiasi nuova entità che venga introdotta in futuro per eventuali modifiche o espansioni del progetto. Questo rende il sistema scalabile e monitorabile, poiché le classi C# stesse offrono una forma di **modello concettuale** supportato dalle **Fluent API**. Tutto questo è molto interessante e ben spiegato, ma manca ancora un aspetto cruciale: l'implementazione concreta di queste tabelle nel database. Potremmo creare manualmente le tabelle tramite

statement SQL che corrispondano agli oggetti C# appena definiti, ma questa soluzione non sarebbe ottimale, soprattutto considerando l'ampiezza dei dati coinvolti, che va ben oltre il semplice sottoinsieme esaminato in questo contesto. È proprio per rispondere a questa esigenza che entrano in gioco le **Migrazioni**.

3.4.2 Migrazioni

Per apportare modifiche alla struttura del database, come l'aggiunta o la rimozione di tabelle o la modifica delle tabelle esistenti, è necessario utilizzare il **DDL (Data Definition Language)**, ovvero quella parte del linguaggio SQL che gestisce il design delle tabelle tramite tali operazioni come `CREATE`, `ALTER`, `DROP`, `TRUNCATE` e altre ancora. Per semplificare queste operazioni, EF Core mette a disposizione le **Migrations**, che consentono non solo di modificare il database con pochi semplici passaggi, ma anche di mantenere la struttura del database sincronizzata con le entità C# definite. Le migrazioni funzionano nel modo seguente: quando viene introdotta una modifica al modello di dati, lo sviluppatore utilizza gli strumenti di EF Core per aggiungere una migrazione che descrive gli aggiornamenti necessari per allineare lo schema del database. EF Core confronta il modello corrente con uno Snapshot del modello precedente per determinare le differenze e genera file di migrazione, i quali possono essere gestiti tramite il controllo del codice sorgente come qualsiasi altro file. Questo sistema ricorda il metodo del controllo di versione: anziché monitorare le versioni di un singolo file, si tiene traccia delle modifiche apportate a un insieme di entità. Dopo questa fase si andrà ad applicare la nuova migrazione al database per rendere effettive le modifiche. Vediamo come sono stati implementati questi due passaggi nel programma di manutenzioni.
Per poter creare una migrazione la prima cosa da fare è quella di installare il pacchetto **Nuget** `Microsoft.EntityFrameworkCore.Design` e anche il comando dotnet per creare le migrazioni (Usando `dotnet tool install -global dotnet-ef`). Grazie a questo comando viene reso disponibile un potente strumento a riga di comando per creare le migrazioni. Il procedimento per creare una migrazione prevede, l'accesso tramite riga di comando alla cartella che contiene il file *.csproj e l'esecuzione del comando `dotnet ef add migrations {NomeMigrazione}`. Vediamo di seguito un esempio di migrazione.

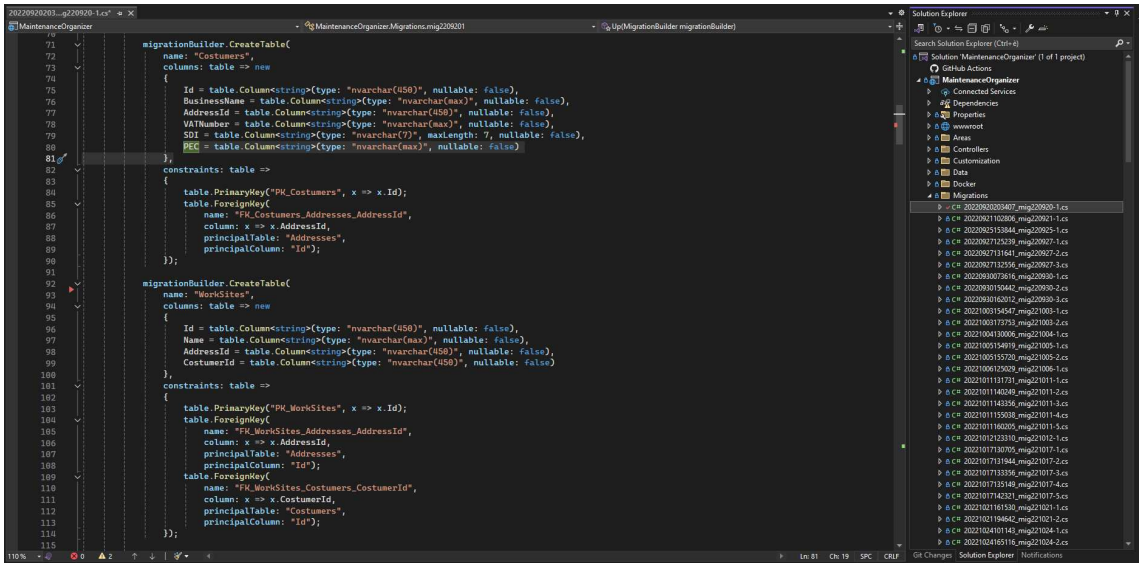


Fig. 18. Esempio migrazione

Nel file `.cs` vengono descritte le operazioni da eseguire sul database utilizzando gli oggetti C# per ottenere i modelli corrispondenti appena creati (come **Customer** e **Worksite**) e vengono anche definite le relazioni esistenti tra di essi. Questo file va poi ad aggiornare un altro file `.cs` nella stessa cartella, il **ModelSnapshot**, che tiene traccia del modello corrente dell'applicazione. Lo screen attuale mostra solo il metodo **Up**, il quale viene eseguito quando si applica la migrazione al database, ma in una migrazione è anche definito il metodo `Down`, che serve a revocare le modifiche apportate da **Up** nel caso in cui si desideri tornare a una versione precedente del model-

lo. L'ultimo passaggio consiste nell'applicare la migrazione al database, e questo può essere fatto seguendo due approcci principali:

- **Comando dotnet**.
  Se si sceglie questa strada, sarà sufficiente applicare il comando `dotnet ef database update {NomeMigrazione}`. Il database di riferimento viene determinato dalla stringa di connessione associata al `DbContext`; in questo caso, essendoci una sola stringa di connessione, verrà utilizzata automaticamente l'unica presente. Tuttavia, questo comando presenta un'inefficienza: richiede l'aggiornamento manuale della struttura del database ogni volta, il che può risultare scomodo, specialmente quando, come nel caso del programma di manutenzione, si hanno due database separati, uno per il test e uno per la produzione. In tale scenario, se si utilizzasse il comando per aggiornare la struttura del database, al momento del passaggio in produzione, il database di produzione potrebbe non essere sincronizzato con il modello attuale, causando errori. Per ovviare a questo problema, entra in gioco la seconda modalità che utilizza il codice C#.

- **Codice C#**.
  Con questa modalità, le migrazioni vengono applicate autonomamente da uno script C# senza l'utilizzo di un comando esplicito. Lo script in questione viene eseguito all'interno del file **Program.cs**, cosi ogni volta che si effettua l'esecuzione dell'applicazione per la prima volta le migrazioni pendenti vengono applicate. Questo approccio non solo risolve il problema descritto in precedenza, ma consente anche di eseguire ulteriori operazioni di modifica sul database qualora le migrazioni da sole non siano sufficienti a implementare le modifiche desiderate. Ad esempio, potrebbe non bastare utilizzare solo le migrazioni in presenza di dati con vincoli particolari, oppure potrebbe essere necessario correggere errori su alcuni campi causati da bug nel programma.

Queste 2 strade sono state entrambe utilizzate nel programma di manutenzioni e una volta visti i numerosi vantaggi del secondo si è optato per l'utilizzo di solo questa procedura. Ecco che con pochi passaggi anziché creare manualmente codice SQL si è aggiornata la struttura del database e si è cosi pronti a manipolare e interrogare i dati.

3.4.3 Organizzazione del codice

Prima di esaminare come avvengono l'interrogazione e la manipolazione dei dati nel programma di manutenzioni, è necessario fare alcune premesse per giustificare l'utilizzo di specifici costrutti. In primo luogo, il servizio responsabile della manipolazione dei dati è denominato **Servizio applicativo**, un servizio che, tramite **Dependency Injection**, riceve l'istanza Scoped del DbContext e la utilizza per eseguire operazioni sul database. Questa scelta di centralizzare le operazioni di manipolazione dei dati in un servizio separato, anziché direttamente nei Controller o nelle Razor Pages, è motivata dalla volontà di mantenere una chiara separazione dei compiti tra i vari componenti del programma, in linea con il principio di modularità.
Un altro aspetto importante riguarda l'uso delle entità all'interno del programma da questo punto in avanti. Le classi C# che rappresentano le entità, descritte in precedenza, non verranno utilizzate direttamente nei Controller o nelle Razor Pages per garantire una chiara separazione dei compiti anche per quanto riguarda la gestione dei dati. Si fa un esempio per chiarire: quando si desidera estrarre una lista di clienti, è evidente che nella tabella HTML non verranno visualizzati tutti i campi della classe **Customer**, ma solo quelli più rilevanti. L'utilizzo diretto della classe **Customer** risulterebbe inefficiente, poiché verrebbero caricati in memoria dati non necessari, aumentando l'overhead e la complessità. Per questa ragione, si ricorre a un **ViewModel**, una classe che rappresenta un sottoinsieme dei campi esposti da **Customer**, utilizzata esclusivamente per la visualizzazione dei dati. Per lo stesso concetto, le operazioni di salvataggio, modifica o eliminazione dei dati si utilizza un **EditInputModel**, una classe che permette di aggiungere o rimuovere informazioni utili solo ai fini della manipolazione dei dati, ma non necessarie per il salvataggio o la mappatura al database. Inoltre, l'**EditInputModel** consente di definire la logica di validazione dei form, un argomento che verrà trattato nella sezione 3.5.

Un altro concetto fondamentale in Entity Framework Core è quello del **tracking** [24] delle entità, ovvero la capacità di tenere traccia delle modifiche apportate alle entità stesse. Questo significa che le modifiche alle entità non vengono immediatamente riflettute nel database quando si assegnano nuovi valori utilizzando l'operatore "=". Le modifiche effettive vengono applicate al database solo quando viene richiamato il metodo `SaveChanges` del **DbContext**. Questo metodo rileva le entità che sono state modificate e le salva nel database. Questo approccio consente di decidere quando salvare i dati nel database, offrendo la possibilità di eseguire operazioni di validazione e sanitizzazione dei dati prima di salvarli, evitando così potenziali errori.
Infine, vorrei commentare la modalità con cui vengono effettuate le operazioni sul database. Queste operazioni sono notoriamente dispendiose in termini di risorse e thread, e se gestite in modo inefficiente possono risultare estremamente lente. Come si può quindi evitare di aspettare che il database completi le sue operazioni di gestione dei dati, riducendo al minimo il tempo perso? La soluzione consiste nell'utilizzare operazioni "asincrone", il cui obiettivo principale è ottimizzare l'uso delle risorse disponibili piuttosto che ridurre il tempo di esecuzione, come avviene tipicamente con le operazioni asincrone tradizionali. In questo modo, il thread che ha avviato l'operazione può svincolarsi dall'attesa e continuare a svolgere altri compiti.
In quali situazioni è consigliabile utilizzare operazioni asincrone? Un esempio tipico è l'interazione con periferiche di I/O, come l'invio di query a un database (esattamente il nostro caso), poiché i protocolli di rete introducono latenza, soprattutto quando si accede a un database remoto. Anche la lettura da disco può beneficiare dell'asincronia, a seconda della velocità del disco, delle dimensioni dei file e delle componenti in gioco. Altri esempi includono l'invio di una richiesta a un web service o l'invio di una mail. Al contrario, non ha senso usare l'asincronia per operazioni che utilizzano intensivamente la CPU, come calcoli numerici, ordinamento di liste o ridimensionamento di immagini, dove il thread resta comunque impegnato attivamente.
L'operatore `async` contrassegna i metodi come asincroni, permettendo loro di eseguire operazioni in modo non bloccante. All'interno di un metodo asincrono, la parola chiave `await` viene utilizzata per sospendere temporaneamente l'esecuzione del flusso di istruzioni fino al completamento dell'operazione asincrona. Questo consente al thread che ha avviato l'operazione di non rimanere inattivo, ma di essere disponibile per ASP.NET Core per svolgere altre attività nel frattempo. Una volta completata l'operazione asincrona, ASP.NET Core può riprendere l'esecuzione del codice, utilizzando un nuovo thread o riutilizzando quello precedente. È importante notare che quando si utilizza `await` in un metodo, è necessario specificare l'attributo `async` nell'intestazione del metodo, in modo da informare il compilatore che il metodo contiene operazioni asincrone. ASP.NET Core consente di utilizzare sia la metodologia asincrona appena vista, in cui si riutilizzano i thread senza ottenere un notevole guadagno in termini di tempo ma un notevole risparmio in termini di risorse, sia la gestione classica delle richieste asincrone, in cui vengono eseguite più operazioni contemporaneamente; in questo caso si deve omettere la parola chiave `await` e far eseguire la singola operazione ad un nuovo thread. Questa struttura organizza l'applicazione come una sorta di "macchina a stati" (**State Machine** [25]), che sospende l'esecuzione di un metodo o di un workflow per eseguire altre operazioni durante l'attesa del completamento dell'operazione asincrona. Quando è necessario riprendere l'esecuzione del codice sospeso, lo stato cambia nuovamente e si passa alla fase di "running".
Questo modello asincrono consente ad ASP.NET Core di gestire un gran numero di richieste con un numero limitato di thread, migliorando l'efficienza complessiva del sistema. I thread non restano bloccati in attesa del completamento delle operazioni asincrone, ma possono passare ad altre richieste, rendendo il server più reattivo. Infine, i metodi che utilizzano `async` nella firma devono restituire un `Task<T>`, per riflettere lo stato del metodo stesso, visto che esegue operazioni asincrone (il termine "stato" è pertinente proprio per il concetto di macchina a stati). Dopo aver introdotto questi preconcetti fondamentali per i prossimi argomenti, si può passare a come avviene la manipolazione e interrogazione dei dati con EF Core.

3.4.4 Querying con LINQ

LINQ (Language Integrated Query) [22] è un framework messo a disposizione dalla piattaforma **.NET** che permette di effettuare delle query sui dati in modo semplice e intuitivo, utilizzando un sintassi simile a SQL direttamente all'interno del codice C#. Le fonti di dati che LINQ può trattare sono molteplici e non solo legate ai

database, come viene anche raffigurato.

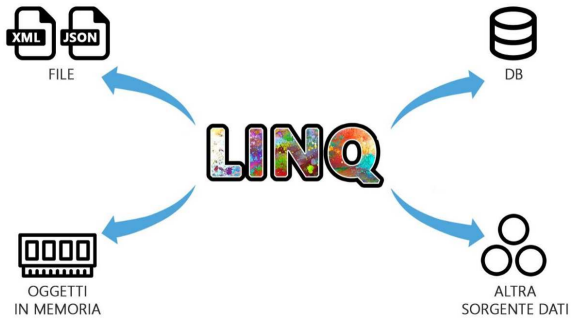


Fig. 19. Proprietà LINQ

La sintassi LINQ è estramente vasta e la sua trattazione non è l'obiettivo della relazione, di conseguenza verrà esaminato un caso d'uso del programma di manutenzioni.

```
3 references
public async Task<List<CustomerViewModel>> GetCustomersListItemsAsync(string customerTypeId, bool onlyCensable,  int skip, int take, string search)
{
    return await DbContext.Customers
        .AsNoTracking()
        .Where(c => (String.IsNullOrEmpty(customerTypeId) ? true : customerTypeId.Equals(c.TypeId)) &&
                    c.ValidItem == true &&
                    (onlyCensable ? c.Censable == true : true) &&
                    (String.IsNullOrEmpty(search) || c.BusinessName.Contains(search)))
        .Include(c => c.Address)
        .Select(c => new CustomerViewModel
        {
            Id = c.Id,
            BusinessName = c.BusinessName,
            Censable = c.Censable,
            HouseNumber = String.IsNullOrEmpty(c.Address.StreetNumber) ? "-" : c.Address.StreetNumber,
            City = String.IsNullOrEmpty(c.Address.City) ? "-" : c.Address.City,
            StringAddress = String.IsNullOrEmpty(c.Address.StreetAddress) ? "-" : c.Address.StreetAddress
        })
        .OrderBy(c => c.BusinessName)
        .Skip(skip)
        .Take(take)
        .ToListAsync();
}
```

Fig. 20. Esempio query LINQ

In questo esempio, la tracciabilità delle entità estratte è stata disabilitata tramite il metodo `AsNoTracking`, poiché l'intento è solo visualizzare i dati senza modificarli. Successivamente, è stato applicato un filtro con `Where` basato sui parametri passati, e una `Join` utilizzando il metodo `Include`, che permette di caricare gli oggetti correlati di interesse. L'intero risultato è stato quindi mappato in entità di tipo **CustomerViewModel** per le ragioni discusse in precedenza. Infine, sono stati applicati ulteriori filtri con `OrderBy` e operazioni di paginazione con `Skip` e `Take`. (La spiegazione dettagliata di questi meccanismi di paginazione sarà completata nella sezione 3.5, dedicata alle **Datatable**). Questa estrazione di dati può essere utilizzata anche per semplici dati caricati in memoria, come per esempio `List<T>` o dei file appena letti. Procediamo ora a vedere come vengono salvati i dati sempre all'interno dello stesso servizio di **Customer**.

3.4.5 Gestione dei dati

Per il salvataggio dei dati nel database vengono utilizzati i metodi forniti dalle collezioni di dati di C#, con l'accortezza di estrarre correttamente il dato dal database, se già presente, e apportare le necessarie modifiche tramite il metodo **SaveChangesAsync**. Nella **Fig. 21** (pagina successiva), si può osservare come inizialmente vengano effettuati controlli di validazione sui dati ricevuti; successivamente, l'entità viene estratta dal database se già esistente, oppure viene creata e aggiunta alla lista dei clienti (senza ancora salvare la modifica nel database). Infine, i valori dei campi vengono aggiornati con quelli provenienti dal **CustomerEditInputModel**, e vengono eseguite operazioni di controllo sull'indirizzo associato. Per salvare queste operazioni, è stata utilizzata l'istruzione `await DbContext.SaveChangesAsync()`, che garantisce un salvataggio dei dati seguendo la logica precedente-

mente discussa. Per eliminare i dati, solitamente si eseguono prima dei controlli sulle eventuali sottoentità associate (ad esempio, un cliente potrebbe avere più **WorkSite** già salvati nel database) e in base alla logica adottata, si possono mostrare messaggi di errore o eccezioni se la cancellazione non può procedere. Se invece è possibile eliminare l'entità, questa viene estratta nello stesso modo descritto in **Fig. 21** per la modifica, e si invoca l'istruzione `DbContext.Customers.Remove(Customer)`, seguita dal consueto `SaveChangesAsync` per salvare la modifica nel database.

```
2 references
public async Task<string> SaveCustomer(CustomerEditInputModel customerEditInputModel)
{
    //Trim del nome del customer
    customerEditInputModel.BusinessName = customerEditInputModel.BusinessName.Trim();

    Customer customer = null;

    int sameNames = DbContext.Customers
        //.AsNoTracking()
        .Where(e => !e.Id.Equals(customerEditInputModel.Id) &&
                    e.BusinessName.Equals(customerEditInputModel.BusinessName) &&
                    (String.IsNullOrEmpty(e.VATNumber) ? true : e.VATNumber.Equals(customerEditInputModel.VATNumber)))
        .Count();

    if( sameNames == 0 )
    {
        if (String.IsNullOrEmpty(customerEditInputModel.Id)) //New
        {
            customer = new Customer()
            {
                ValidItem = true,
            };

            DbContext.Customers.Add(customer);
        }
        else //Edit
        {
            customer = await DbContext.Customers
             .Where(c => c.Id.Equals(customerEditInputModel.Id))
             .Include(c => c.Address)
             .FirstOrDefaultAsync();

        }

        customer.BusinessName = customerEditInputModel.BusinessName;
        customer.Censable = customerEditInputModel.Censable;
        customer.VATNumber = customerEditInputModel.VATNumber;
        customer.Email = customerEditInputModel.Email;
        customer.PhoneNumber = customerEditInputModel.PhoneNumber;
        customer.TypeId = String.IsNullOrEmpty(customerEditInputModel.CustomerTypeId) ? null : customerEditInputModel.CustomerTypeId;
        customer.SDI = customerEditInputModel.SDI;
        customer.PEC = customerEditInputModel.PEC;

        bool validAddress = AddressService.IsValid(customerEditInputModel.Address);

        if (validAddress)
        {
            if (customer.Address == null)
            {
                customer.Address = new Address();
            }
            //customer.Address = customerEditInputModel.Address;

            customer.Address.State = customerEditInputModel.Address.State;
            customer.Address.StreetAddress = customerEditInputModel.Address.StreetAddress;
            customer.Address.CAP = customerEditInputModel.Address.CAP;
            customer.Address.City = customerEditInputModel.Address.City;
            customer.Address.Province = customerEditInputModel.Address.Province;
            customer.Address.StreetNumber = customerEditInputModel.Address.StreetNumber;
            customer.Address.Latitude = customerEditInputModel.Address.Latitude;
            customer.Address.Longitude = customerEditInputModel.Address.Longitude;

        }
        else
        {
            if (customer.Address != null)
            {
                DbContext.Addresses.Remove(customer.Address);
                customer.Address = null;
            }
        }

        await DbContext.SaveChangesAsync();

        //Inalido la cache
        MemoryCache.Remove(CacheKeys.CUSTOMERS_LIST);


        return customer.Id;
    }
    else
    {
        //throw new ArgumentException("Cannot insert duplicate names");
        return null;
    }
}
```

Fig. 21. Esempio di salvataggio di customer

Ora che le principali operazioni di manipolazione dei dati sono state trattate, è possibile procedere nel parlare del lato **Front end** e di come questo viene gestito in ASP.NET Core.

3.5 Visualizzazione dei dati

Fino a questo punto si è discusso di come progettare, organizzare e manipolare i dati. Ora vediamo come raccoglierli e visualizzarli utilizzando gli strumenti di gestione della UI offerti da ASP.NET Core. Come già accennato, per la gestione delle pagine web è stata scelta la tecnologia delle **Razor Pages**, in quanto offre una migliore organizzazione della struttura delle cartelle, un potente meccanismo di data-binding e ricorda una tecnologia precedente e familiare all'azienda, ovvero ASP.NET Web Forms. Una parte della struttura delle pagine che il sito presenta è stata organizzata in cartelle presenti al livello più alto (E.g Pages, Views e ViewComponent). Mentre, un'altra parte è organizzata all'interno di una cartella speciale che prende il nome di **Areas**: Questa cartella permette di suddividere le varie cartelle per macroconcetti rendendo lo sviluppo più ordinato. Le aree (o cartelle) che sono state sviluppate per il programma di manutenzioni sono 3:

- **Admin**.
  In quest'area sono presenti tutte le anagrafiche e funzionalità che possono utiizzare gli admin che gestiscono questa piattaforma. In questa sezione ci si occupa della gestione dell'utenza, cioè creare nuovi utenti, assegnare dei ruoli o togliere dei permessi.

- **Identity**.
  Questa area viene creata automaticamente dal framework **Identity** [6], grazie all'operazione di scaffolding. Questo insieme di pagine riguarda la parte di gestione del proprio account comprese anche le pagine di login, recupero password, registrazione e molte altre. Inoltre, essendo delle pagine Razor sono assolutamente personalizzabili dal punto di vista grafico, mentre dal punto di vista del funzionamento si vanno a fare solo poche modifiche mirate visto che Microsoft ha progettato nel dettaglio un framework che si occupi in modo efficente della gestione dell'utenza.

- **User**.
  Questa area è di particolare interesse per questa sezione, poiché riguarda tutte le pagine accessibili agli utenti del sito. Qui si trovano le pagine relative alle anagrafiche e alla gestione delle manutenzioni, con le varie funzionalità di ciascuna pagina regolate dal meccanismo dei permessi (questo aspetto sarà trattato nella sezione successiva).

Infine, vorrei sottolineare che le pagine di utilizzo globale, come l'**Index**, la pagina dell'informativa sulla privacy, la pagina degli errori e il layout, sono tutte definite all'interno della cartella **Pages**, situata allo stesso livello del file `*.csproj`. È importante soffermarsi anche sul concetto di layout che ASP.NET Core mette a disposizione. Il **layout** [26] è una pagina modello che definisce la struttura visiva comune e condivisa tra più pagine web in un'applicazione; consente di centralizzare elementi come la sidebar, il footer e la barra di navigazione, che vengono utilizzati in tutte le pagine dell'applicazione, riducendo così la ripetizione del codice. Le pagine individuali possono inserire il proprio contenuto dinamico all'interno di specifiche sezioni del layout, mantenendo coerenza e uniformità nell'interfaccia utente dell'applicazione. Nel programma in questione, è stato creato un file di layout per definire la side bar e la navbar. Ogni pagina Razor viene renderizzata in modo tale che il layout completo venga mostrato, con l'aggiunta del contenuto della specifica pagina Razor. Per poter inizializzare correttamente un layout, è necessario configurare un file speciale chiamato `_ViewStart.cshtml`.
In generale, una Razor View (file `.cshtml`) è una pagina HTML che può contenere sia codice HTML che codice C#. Questa combinazione permette di generare dinamicamente il codice HTML utilizzando costrutti C# (ad esempio, eseguire un ciclo `for` per creare 10 elementi `div` contenenti la scritta "Hello World!"). Oltre alla possibilità di creare HTML dinamico in modo semplice e senza l'uso di JavaScript, ASP.NET Core consente di rendere la Razor View **fortemente tipizzata**, grazie alla direttiva `@model`. Questo significa che è possibile definire un modello di riferimento (file `.cshtml.cs`) per la Razor View corrispondente, abilitando operazioni di data-binding per inizializzare entità che verranno aggiunte al database (vedi **Fig. 23**, **Fig. 24** e **Fig. 25**, con la spiegazione a pagina 41).
Un altro componente molto utile in ASP.NET Core è il **Tag Helper** [27]. I **Tag Helper** [27] sono apparentemente

dei semplici tag HTML, ma in realtà, grazie a proprietà e attributi appositi, questi vengono elaborati dal **View Engine** e arricchiti di altri attributi che normalmente non avremmo (e.g., `asp-for`, `asp-page`, `asp-area`, `asp-page-handler`, e molti altri). Uno dei vantaggi dei **Tag Helper** è quello di lasciare la view molto leggibile nonostante si carichi di un po' di logica applicativa e la possibilità di definire dei Tag Helper personalizzati per far fronte a delle esigenze che non vengono soddisfatte dai Tag Helper predefiniti di Microsoft. Per poter inizializzare correttamente i Tag Helper, è necessario configurare un file speciale chiamato `_ViewImports.cshtml`, che contiene gli `using` dei namespace utilizzati dalle Razor View.
Passiamo ora a vedere un esempio di Razor View utilizzata per la definizione di un form di inserimento di un cliente. Il risultato finale ottenuto è mostrato di seguito. È importante notare che, per facilitare lo sviluppo dei vari componenti HTML, sono state utilizzate anche librerie grafiche specifiche, che hanno contribuito a migliorare l'aspetto dell'app.

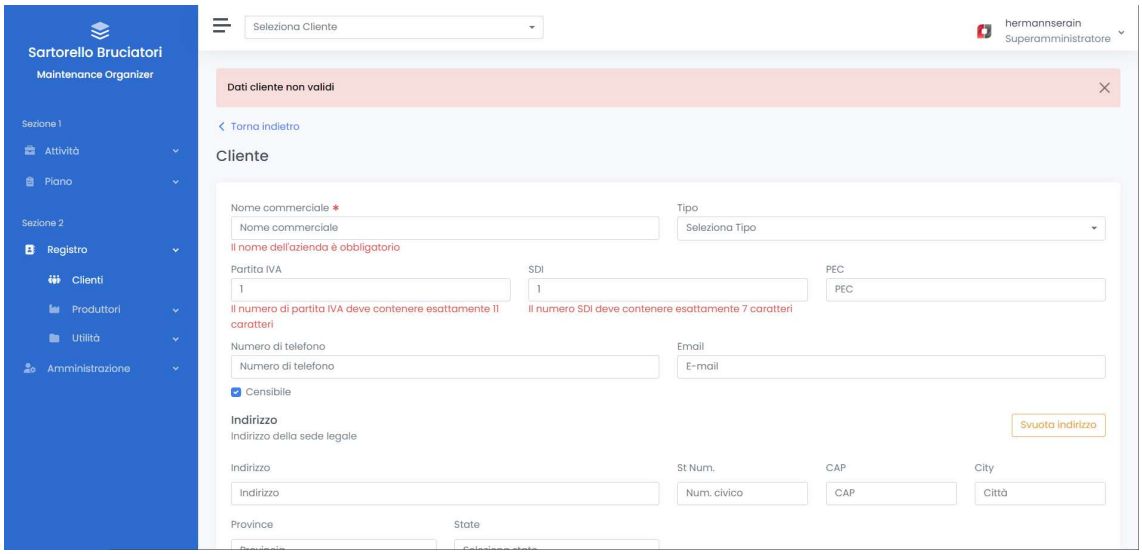


Fig. 22. Pagina di inserimento cliente con errori di validazione

In **Fig. 22** la pagina Razor è costituita solo da ciò che ha lo sfondo in grigio. La navabar in alto e la side bar a sinistra fanno parte del layoyt predefinito. Di seguito alcune parti del file *.cshtml.cs e *.cshtml di riferimento per comprendere il lavoro svolto.

```
@page "{id?}"
@model MaintenanceOrganizer.Areas.User.Pages.Customers.EditModel
@{
    string pageTitle = localizer["Customer"];
    //ViewData["Title"] = pageTitle;
    ViewData["Title"] = Model.CustomerEditInputModel.BusinessName;
    ViewData["SideBarMainItem"] = "Registry";
    ViewData["SideBarSubItem"] = "CustomersRegistry";
}

<div class="d-flex flex-row">...

<div class="d-flex flex-row mb-3">
    <div class="col-auto d-none d-sm-block">
        <h3>@localizer[pageTitle]</h3>
    </div>
</div>

<div class="col-md-12">
    <form method="post">
        <div class="card">
            <div class="card-body">
                <div class="row mt-1">
                    <div class="mb-2 col-md-6">
                        <label class="form-label mb-0" asp-for="@Model.CustomerEditInputModel.BusinessName"></label>
                        <label><i class="fa-solid fa-fw fa-star-of-life fa-2xs text-danger"></i></label>
                        <input type="text"  asp-for="@Model.CustomerEditInputModel.BusinessName" class="form-control" placeholder="@localizer["Business Name"]">
                        @* <span asp-validation-for="@Model.CustomerEditInputModel.BusinessName"></span> *@
                        @Html.ValidationMessageFor(Model => Model.CustomerEditInputModel.BusinessName, "", new { @class = "text-danger" })
                    </div>
```

Fig. 23. Estratto di Razor View

```
[Authorize(Policy = PolicyUtility.EDITABLEANDVIEWABLE_REGISTRY_POLICY)]
6 references
public class EditModel : PageModel
{
    0 references
    public EditModel(
        IStringLocalizer localizer,
        EFCustomerConfigurationService customerConfigurationService,
        EFCustomerService customerService,
        EFAddressService addressService)
    {
        Localizer = localizer;
        CustomerConfigurationService = customerConfigurationService;
        CustomerService = customerService;
        AddressService = addressService;
    }

    [BindProperty(SupportsGet = true)]
    55 references
    public CustomerEditInputModel CustomerEditInputModel { get; set; }
    4 references
    public EFCustomerService CustomerService { get; }
    6 references
    public IStringLocalizer Localizer { get; }
```

Fig. 24. DI nel Razor Model

```
0 references
public async Task<IActionResult> OnPostSaveAsync()
{
    if (ModelState.IsValid)
    {
        CustomerEditInputModel.Address = AddressService.IsValid(CustomerEditInputModel.Address) ? CustomerEditInputModel.Address : null;
        //Nome cliente univoce, possibile eccezzione da gestire
        try
        {
            CustomerEditInputModel.Id = await CustomerService.SaveCustomer(CustomerEditInputModel);
            TempData["ConfirmationMessage"] = Localizer["Customer saved succesfully"].Value;
            //TempData["ConfirmationMessage"] = "Customer saved succesfully";
            return RedirectToPage("/customers/edit", new { Id = CustomerEditInputModel.Id, ReturnUrl = ReturnUrl });

        }
        catch (Exception ex)
        {
            string msg = ex.InnerException?.Message ?? ex.Message;
            if (msg.Contains("Cannot insert duplicate key row"))
            {
                TempData["ErrorMessage"] = Localizer["This customer name already exists"].Value;
                //TempData["ErrorMessage"] = "This customer name already exists";
                ModelState.AddModelError("CustomerEditInputModel.BusinessName", Localizer["This name already exists"].Value);
                return Page();
            }
            else
            {
                throw ex;
            }
        }
    }
    else
    {
        TempData["ErrorMessage"] = Localizer["Customer data invalid"].Value;
        await LoadEntities(true);
        return Page();
    }

}
```

Fig. 25. Page handler

La prima figura da considerare è la **Fig. 24**, dove vengono iniettati i servizi applicativi necessari tramite la **Dependency Injection**. Successivamente, viene definita la proprietà `CustomerEditInputModel`, che ha il compito di contenere i dati di un nuovo cliente da inserire o di un cliente esistente da modificare. È importante segnalare che questa proprietà deve supportare il data-binding, per poter mappare correttamente i valori dei tag HTML nell'oggetto C#. Questa operazione viene svolta tramite la Data Annotation `[BindProperty(SupportsGet = true)]`. Nella **Fig. 23** viene illustrato un form che utilizza il tag-helper `input`, dove l'attributo speciale `asp-for` effettua la mappatura tra l'HTML e l'attributo dell'oggetto C#, per poter fare il binding di tutti gli attri-

buti dell'oggetto `CustomerEditInputModel`, basterà creare tanti input quanti sono gli attributi. Sotto ogni input è presente un'istruzione che visualizza eventuali messaggi di errore (**Fig. 22**) basati sulla validazione del form. Questa validazione è definita tramite le Data Annotation presenti nella classe `CustomerEditInputModel` (**Fig. 26**), le quali specificano se un attributo è obbligatorio, la lunghezza massima, il tipo di formattazione e altri vincoli possibili. Il controllo effettivo della validazione avviene nel **page handler** illustrato in **Fig. 25**. Il **page handler** rappresenta il metodo di destinazione del form al momento del click del pulsante "Salva" (**Fig. 22**), ovvero il pulsante di submit. A questo pulsante è associato un **page handler** specifico, nel nostro caso quello di salvataggio. Tornado alla **Fig. 25** se i vincoli definiti non vengono rispettati, l'esecuzione non entra nel `if` e viene restituita la pagina con degli errori; se invece tutti i dati vengono inseriti correttamente, viene eseguito il metodo **SaveCustomer**, già discusso in **Fig. 21**.
Questo sistema di creazione di un form consente di sviluppare anagrafiche molto articolate e ben organizzate, poiché ogni file ha un compito specifico e ben definito. Seguendo gli esempi presentati e le operazioni descritte, è possibile continuare a definire pagine per l'inserimento, la modifica e la cancellazione di tutte le entità presentate in precedenza, come **Worksite**, **IndustryPlant**, **Element** e altre. Nei prossimi paragrafi vedremo alcuni esempi di implementazione di una tabella paginata con filtro di ricerca e una mappa per la gestione degli indirizzi delle sedi.

```
16 references
public class CustomerEditInputModel
{
    //[FromRoute(Name = "customerid")]
    30 references
    public string? Id { get; set; }

    [Required(ErrorMessage = "The Business Name is required"),
    MinLength(2, ErrorMessage = "The Business Name must be at least {1} charachters long"),
    MaxLength(200, ErrorMessage = "The Business Name must be maximum {1} charachters long"),
      RegularExpression(@"^[0-9A-z\u00C0-\u00ff\s\.']+$", ErrorMessage = "Invalid Business Name"),
    Display(Name = "Business Name")]
    24 references
    public string BusinessName { get; set; }

    [MinLength(11, ErrorMessage = "The VAT Number must be exactly {1} charachters"),
    MaxLength(11, ErrorMessage = "The VAT Number must be exactly {1} charachters"),
    RegularExpression(@"^[0-9]+$", ErrorMessage = "Invalid VAT Number"),
    Display(Name = "VAT Number")]
    19 references
    public string? VATNumber { get; set; }

    [MinLength(7, ErrorMessage = "The SDI Number must be exactly {1} charachters"),
        MaxLength(7, ErrorMessage = "The SDI Number must be exactly {1} charachters"),
    RegularExpression(@"^[A-Za-z0-9]*$", ErrorMessage = "Invalid SDI Number"),
    Display(Name = "SDI")]
    5 references
    public string? SDI { get; set; }
```

Fig. 26. Classe CustomerEditInputModel

3.5.1 Datatable

Una funzionalità cruciale nello sviluppo di anagrafiche è la visualizzazione delle entità in formato tabellare. Questa caratteristica è fondamentale per consentire all'utente di navigare tra gli oggetti memorizzati nella piattaforma, poiché senza di essa sarebbe impossibile ritrovare e modificare le entità inserite nel sistema. Le tipologie di tabelle possono variare da quelle più semplici, che mostrano tutti i dati presenti, a quelle più complesse, dotate di paginazione, ordinamento e filtraggio dei dati. Nel programma in questione, sono state implementate diverse tabelle, ciascuna progettata per soddisfare esigenze specifiche. In questo caso, considerando che la lista dei clienti è molto lunga, mostrare tutti i dati contemporaneamente sarebbe stato inefficace e inutile. Per questo motivo, è stato implementato un sistema di paginazione, filtraggio e ordinamento delle entità, sfruttando la libreria JavaScript **Datatable** [28].

Per utilizzare **Datatable** [28], il primo passo consiste nel creare un file JavaScript da includere nella Razor View contenente la tabella. Successivamente, è necessario inizializzare la tabella datatable all'interno dell'evento **DOMContentLoaded**, specificando i parametri appropriati per implementare le funzionalità desiderate (E.g `pagination: true`). Inoltre, è necessario predisporre un controller accessibile tramite una richiesta POST. Questo può essere fatto utilizzando le chiamate **Ajax** [29], insieme alla Data Annotation `[HttpPost]` per indicare che l'action del controller gestisce esclusivamente richieste POST.
Il controller riceverà parametri che permettono di eseguire ricerche specifiche, e inoltre dovrà accettare un oggetto che mappi tutti i parametri inviati da Datatable per funzionare correttamente. Questi includono informazioni come la pagina corrente, il numero di elementi per pagina e il contenuto della input box di ricerca. Questa mappatura avviene tramite una classe definita dal programmatore, denominata `DataTableInputModel`, che rispetta gli attributi specificati nella documentazione di Datatable. Con questa classe, sarà possibile gestire la paginazione e i filtri menzionati in precedenza.
Una volta che il controller ha ricevuto i parametri, li passerà al servizio applicativo responsabile dell'estrazione dei dati, utilizzando il metodo illustrato in **Fig. 20**. Infine, Datatable consente di definire colonne completamente personalizzabili, una funzionalità che è stata sfruttata per includere icone, come la matita per l'edit e l'occhio per la visualizzazione, delle rispettive pagine. Di seguito l'esempio UI del risultato.

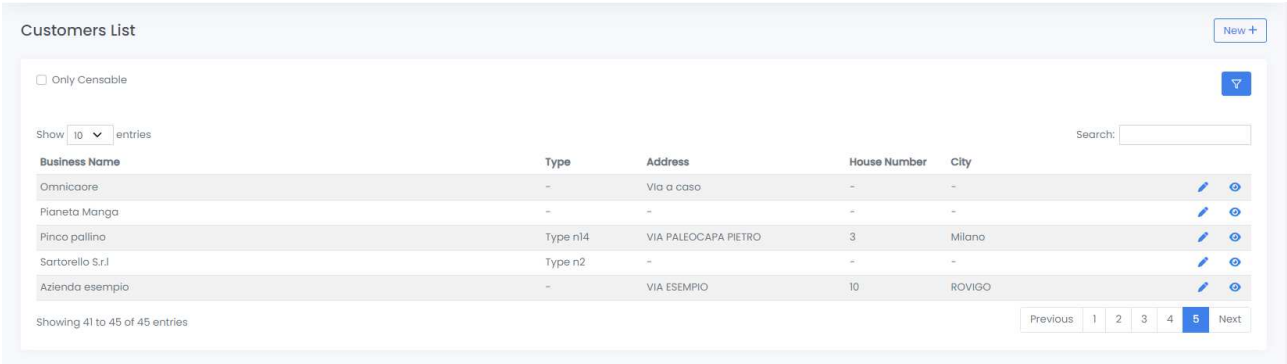


Fig. 27. Lista di clienti con Datatable

3.5.2 Mappe

Un altro esempio fornito nel programma di manutenzioni è lo sviluppo di una mappa in grado di mostrare la posizione di una sede del cliente in base al suo indirizzo. Per realizzare la mappa è stata utilizzata la libreria open source **Leaflet** [30] che permette l'utilizzo di layer gratuiti come **OpenStreetMap**. L'inizializzazione della mappa, per quanto riguarda la parte JavaScript, segue un procedimento simile a quello utilizzato per la tabella Datatable. Successivamente, in base alla documentazione, sono stati definiti i parametri per mostrare una posizione di default (nel nostro caso, Rovigo) e implementare diverse funzionalità, come la possibilità di aggiornare la latitudine e la longitudine muovendo il marker rosso, o viceversa, spostare il marker rosso in base all'aggiornamento delle coordinate. Cliccando il pulsante "Get Coord. From Address" (**Fig. 28**) vengono calcolate le coordinate a partire dalle informazioni fornite nei campi dell'indirizzo, con conseguente aggiornamento della mappa. Questa operazione è resa possibile grazie a un'API specifica messa a disposizione da **Geoapify** [31]. Inoltre, il pulsante "Get Directions" consente di aprire l'applicazione Google Maps per indicare la destinazione, una funzione pensata per essere utilizzata anche su tablet, facilitando così i manutentori nella localizzazione delle sedi per le manutenzioni. Infine, è stata aggiunta la possibilità di calcolare metriche di distanza e tempo stimato dalla posizione della sede alla posizione attuale dell'**OwnerSettings**, una funzionalità resa possibile grazie alle API fornite da Google.
Di seguito il risultato ottenuto:

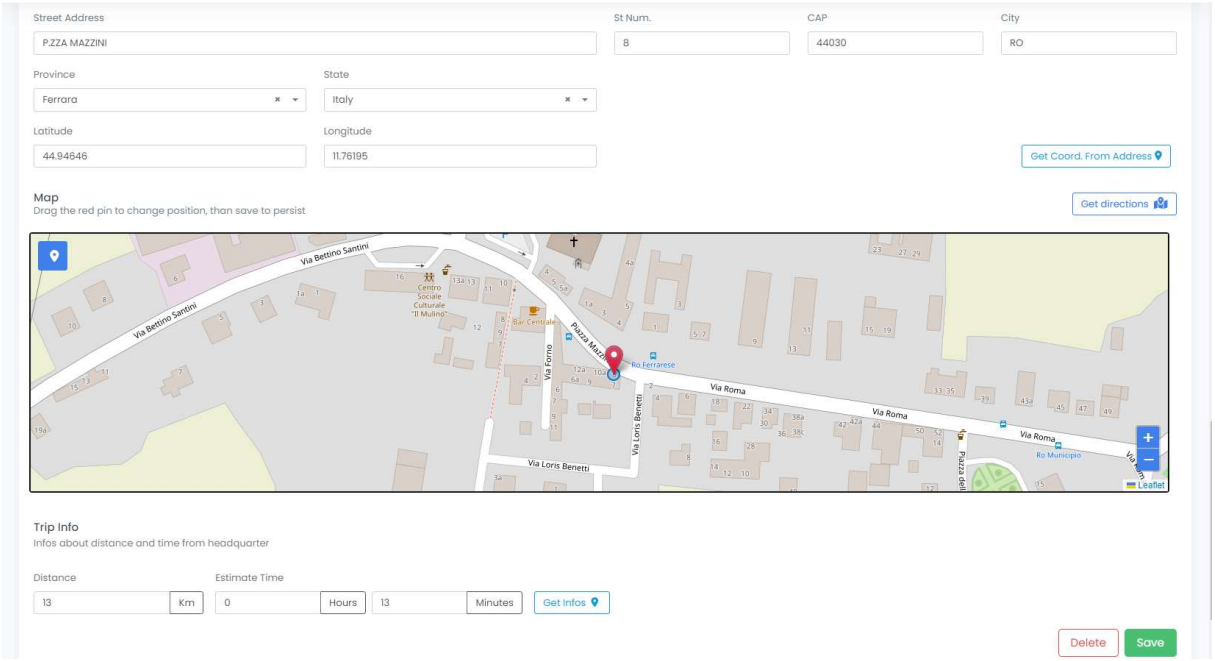


Fig. 28. Mappa dell'indirizzo delle sedi

3.6 Autenticazione e Autorizzazione

L'ultimo argomento trattato in questa relazione riguarda la sicurezza dell'applicazione, un aspetto cruciale poiché, allo stato attuale, l'accesso è aperto a tutti, il che naturalmente non è ammissibile. Verranno esplorati quindi i concetti fondamentali attraverso i quali ASP.NET Core assicura la protezione dell'applicazione e analizzeremo come questi meccanismi siano stati implementati nel progetto, sfruttando il framework dedicato alla gestione della sicurezza.

3.6.1 Concetti principali

Come suggerisce il titolo della sezione, i due aspetti principali da garantire sono l'**Autenticazione e l'Autorizzazione**. Vediamo di seguito la differenza tra questi due concetti:

- **Autenticazione**.
  La fase di autenticazione si occupa di determinare l'identità dell'utente basandosi su un documento valido, emesso da un'autorità affidabile e presentato dall'utente stesso. Questo concetto è applicabile in molti ambiti della vita quotidiana; ad esempio, quando ci viene richiesto un documento di riconoscimento per confermare la nostra identità. Un esempio concreto è quando ci si reca allo stadio per un evento sportivo: gli steward, all'ingresso, chiedono un documento d'identità per verificare chi siamo e nel caso di esito positivo si prosegue verso i tornelli per esibire il biglietto.

- **Autorizzazione**.
  La fase di autorizzazione è il passo successivo a quella di **Autenticazione**. Una volta confermata e riconosciuta l'identità dell'utente, è necessario determinare se quest'ultimo possiede i permessi necessari per eseguire l'operazione richiesta. Riprendendo l'esempio precedente, dopo aver superato il controllo degli steward allo stadio, sarà necessario esibire il biglietto al tornello corrispondente. In base al tipo di biglietto acquistato, si potrà accedere a determinate aree: ad esempio, tentare di entrare nella tribuna d'onore con un biglietto per la curva non darà esito positivo, visto che possediamo un biglietto valido per un'altra area dello stadio.

Questi concetti sono gestiti in modo ottimale dal framework **ASP.NET Core Identity**. Questo framework è stato progettato nei minimi dettagli per garantire non solo sicurezza, ma anche una grande scalabilità, rendendolo perfetto per gestire un ampio numero di utenti.

3.6.2 I principi di Identity

ASP.NET Core rappresenta una soluzione ottimale rispetto a una soluzione personalizzata principalmente per il suo elevato livello di astrazione e per l'integrazione che offre con i meccanismi sottostanti. Ad esempio, Identity è indipendente dal tipo di database utilizzato, consentendo di scegliere tra SQL Server, PostgreSQL o qualsiasi altro DBMS. Inoltre, permette di utilizzare diverse tecnologie di persistenza, come EF Core o ADO.NET. Un altro vantaggio di Identity è la possibilità di autenticare utenti già registrati su altre piattaforme come Facebook, Google o Microsoft, evitando così la necessità di creare un nuovo account per ogni sito web. I pacchetti essenziali con cui **Identity** opera sono sostanzialmente quattro, e verranno descritti di seguito.

- **Microsoft.AspNetCore.Identity.UI**
  Come suggerisce il nome, questo pacchetto contiene tutta la grafica predefinita creata da Microsoft per gestire le fasi di registrazione, login, recupero password e altre operazioni correlate. Questi elementi sono accessibili tramite l'operazione di scaffolding di Identity, che include la creazione della relativa **Area**. Inoltre, è possibile personalizzare l'aspetto delle pagine generate per allinearle al design del programma, come è stato fatto nel progetto di manutenzioni. Oltre agli aspetti grafici, il pacchetto gestisce anche i modelli di input necessari per raccogliere le informazioni inserite nei vari form.

- **Microsoft.AspNetCore.Identity.EntityFrameworkCore**
  Quando si sceglie di utilizzare EF Core come meccanismo di persistenza per gestire tabelle relative a utenti, ruoli, claim e altre informazioni correlate, è necessario derivare il proprio DbContext dalla classe **IdentityDbContext<T>**, dove **T** rappresenta la classe che modella i dati dell'utente. Questo approccio è stato adottato anche nel programma in questione, come illustrato nell'esempio mostrato in **Fig. 16**.

- **Microsoft.Extensions.Identity.Store**
  Questo pacchetto, insieme al successivo, costituisce una dipendenza rispetto ai primi due pacchetti. Esso contiene le classi di entità che descrivono il profilo dell'utente, come ad esempio `IdentityUser<TKey>`, utilizzata come classe base per la definizione di `ApplicationUser`, a cui sono stati aggiunti campi specifici relativi al programma.

- **Microsoft.Extensions.Identity.Core**.
  Come suggerisce il nome, questo pacchetto costituisce il nucleo di Identity, contenendo i servizi applicativi come UserManager e RoleManager, che permettono di gestire gli utenti all'interno della piattaforma, ad esempio per estrarre gli utenti registrati.

Dopo aver delineato la struttura di base di Identity, passiamo ora alla gestione concreta di questi concetti all'interno della piattaforma. La prima operazione da effettuare è la **registrazione**, durante la quale l'utente si presenta alla piattaforma fornendo i propri dati, in questo caso semplicemente un'email e una password. Successivamente, si passa alla fase di login, in cui l'utente conferma la propria identità inserendo nuovamente l'email e la password (ovvio che in questo caso non tutti possono registrarsi all'applicazione ma solo un numero ristretto di email). Una volta superata questa fase, viene generato un **cookie di autenticazione**, che Identity utilizza per tenere traccia dell'identità dell'utente durante le varie richieste effettuate nella sessione corrente. È proprio grazie a questo cookie che Identity, all'interno della pipeline di una richiesta HTTP, può verificare se l'utente è autenticato o meno, tramite specifici **middleware** dedicati all'autenticazione e all'autorizzazione. L'ordine di posizionamento di questi middleware è cruciale, poiché se inseriti nel punto sbagliato, risulterebbero completamente inefficaci. Di solito, questi middleware vengono collocati dopo l'endpoint di routing, ovvero dopo che è stato effettuato il mapping dell'URL e si sa quale action o Razor Page verrà invocata. A questo punto, **il middleware di autenticazione viene eseguito per primo** per validare l'identità dell'utente. Successivamente, **il middleware di autorizzazione** viene

eseguito per verificare i permessi che l'utente possiede. In base ai risultati di queste validazioni, la richiesta può essere fermata o fatta proseguire.
Per quanto riguarda il codice, Identity memorizza l'identità dell'utente che ha effettuato la richiesta HTTP corrente all'interno di un oggetto `User` di tipo `ClaimsPrincipal`. Questo oggetto può contenere zero o più `ClaimsIdentity`, che possono essere considerati come documenti d'identità virtuali. Ogni `ClaimsIdentity` contiene una serie di coppie chiave-valore, chiamate "claim", che rappresentano le informazioni dell'utente. Ad esempio, un claim potrebbe contenere le coppie `Name-Mario`, `Cognome-Serain` oppure `Ruolo-admin`. Questo meccanismo consente di accedere alle informazioni dell'utente in tutto il progetto.
Per soddisfare le esigenze del programma di manutenzioni, è necessaria un'ulteriore distinzione tra gli utenti autenticati. Non basta infatti differenziare tra utenti autenticati e non; è fondamentale organizzare i sottoinsiemi degli utenti autenticati in base ai permessi. Ciò significa che alcuni utenti avranno accesso a tutte le operazioni disponibili, mentre altri potranno eseguire solo alcune. Questa gestione dei permessi è resa possibile grazie all'uso dei ruoli.
I ruoli vengono definiti all'avvio del programma se non sono già presenti. Essi rappresentano un insieme di permessi: ad esempio, un amministratore avrà accesso completo a tutte le funzioni, un manutentore potrà visualizzare solo l'anagrafica dei macchinari e un manager avrà la facoltà di gestire tutte le anagrafiche e assegnare ruoli agli utenti. All'avvio dell'applicazione, viene stabilita una lista iniziale di amministratori, definita nel file `appsettings.json`, che si occuperanno di creare nuovi utenti e assegnare loro i ruoli in base alle mansioni specifiche.
Per limitare l'accesso alle pagine dell'applicazione in base ai ruoli, è possibile definire delle policy e proteggere determinate aree, pagine o porzioni di pagine in base a tali policy. Per proteggere specifici controller, si possono utilizzare annotazioni come `[Authorize]`. In questo modo si garantisce che ogni utente possa accedere solo alle funzionalità pertinenti alla sua mansione. Di seguito un esempio di lista di utenti in cui viene mostrata la finestra di assegnazione dei ruoli, da cui si vedono tutti i ruoli presenti all'interno della piattaforma.

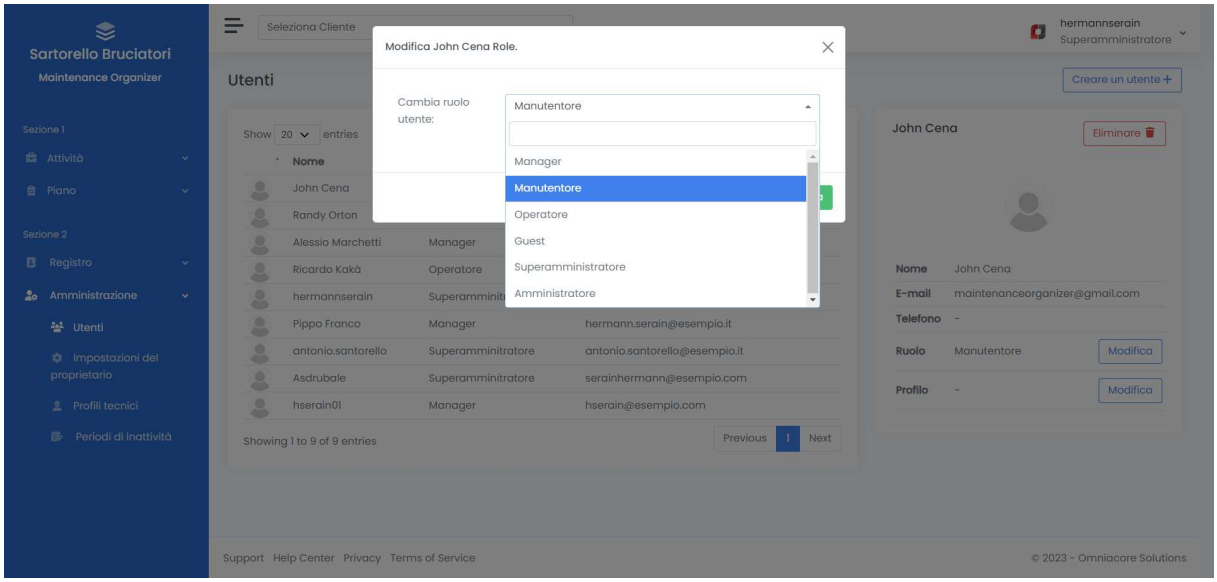


Fig. 29. Lista di utenti con modale per la modifica del ruolo

4 Conclusioni

Lo sviluppo di Web App ha oggi un mercato molto vasto, grazie all'evoluzione digitale degli ultimi decenni. La crescente diffusione di dispositivi digitali ha portato non solo a un utilizzo sempre più esteso di queste tecnologie, ma anche a una transizione digitale che ha trasformato i vecchi sistemi di archiviazione su carta in nuovi gestionali completamente in cloud. Sviluppare una tecnologia che risponda a queste esigenze è estremamente utile nel settore informatico e altamente formativo a livello professionale, considerate la varie tecnologie coinvolte, come ASP.NET Core, Identity, EF Core, Datatable, Leaflet, e molte altre non trattate in questa relazione. Ritengo che questa esperienza lavorativa mi abbia formato sotto molti aspetti. Affrontare per la prima volta la creazione di un progetto da zero mi ha permesso di comprendere meglio l'intero processo di sviluppo di un software, dalle sue fondamenta fino alla pubblicazione e inoltre mi sono ritrovato a dover affrontare numerose tecnologie per la prima volta il che mi ha portato, necessariamente, a migliorare la mia predisposizione al problem-solving, rispetto all'esperienza passata che risultava essere abbastanza limitata. Questo aspetto ha contribuito ad arricchire il mio bagaglio professionale e formativo, il che mi rende molto soddisfatto nell'aver intrapreso questo progetto e mantengo la speranza e la voglia di poterne fare ancora molti altri.

Per quanto riguarda la tecnologia utilizzata, considero ASP.NET Core una delle soluzioni migliori sul mercato, grazie alla chiarezza con cui vengono esposti i concetti e al continuo sviluppo di nuove tecnologie che contribuisce a mantenere ASP.NET Core costantemente aggiornato e competitivo. Un esempio significativo è **Blazor** [8], ovvero una nuova tecnologia per poter migliorare lo sviluppo Web andando a integrare ancor di più con l'utilizzo di codice C# e di architetture per l'ottimizzazione dell'esecuzione di codice, sia lato server che lato client. Un'altra caratteristica fondamentale di questa tecnologia è la sua natura cross-platform, che libera dalla dipendenza dai server Microsoft, consentendo l'esecuzione delle applicazioni su server Linux, la tecnologia predominante nel campo del software server.

Concludo dicendo che per gli aggiornamenti futuri dell'applicazione, oltre alle 225 ore di attività lavorativa, sono state aggiunte ulteriori funzionalità per la gestione degli interventi e dei contratti di lavoro, al fine di definire concretamente le operazioni che costituiscono il cuore dell'applicazione. Inoltre, è stata ampliata l'anagrafica, soprattutto per quanto riguarda la componentistica e la gestione del magazzino dei pezzi di ricambio, per garantire una migliore mappatura delle scorte nelle varie sedi dei clienti.

Sitografia

Link utili:

- **ASP.NET Core per tutti** [1], https://www.udemy.com/course/aspnetcore-per-tutti/
- **Omniacore Solutions** [2], https://omniacore.it/it
- **Upndw** [3], https://upndw.com/it
- **Pubblie** [4], https://pubblie.io/it
- **Wikipedia** [5], https://it.wikipedia.org/
- **Identity** [6], https://learn.microsoft.com/en-us/aspnet/core/security/?view=aspnetcore-8.0
- **Hosting flessibile** [7], https://learn.microsoft.com/it-it/aspnet/core/host-and-deploy/?view=aspnetcore-8.0
- **Blazor** [8], https://learn.microsoft.com/it-it/aspnet/core/blazor/?view=aspnetcore-8.0
- **ML .NET** [9], https://dotnet.microsoft.com/it-it/apps/machinelearning-ai/ml-dotnet
- **Nuget** [10], https://learn.microsoft.com/it-it/nuget/
- **Visual Studio** [11], https://learn.microsoft.com/en-us/visualstudio/windows/?view=vs-2022
- **RCS** [12], https://www.gnu.org/software/rcs/
- **Git** [13], https://git-scm.com/doc
- **Github** [14], https://docs.github.com/en
- **Postman** [15], https://learning.postman.com/docs/introduction/overview/
- **SQL Server Management Studio** [16], https://learn.microsoft.com/it-it/sql/ssms/sql-server-management-studio-ssms?view=sql-server-ver16&source=recommendations
- **Docker** [17], https://docs.docker.com/guides/docker-overview/
- **Traefik** [18], https://doc.traefik.io/traefik/
- **CLI** [19], https://learn.microsoft.com/en-us/dotnet/core/tools/
- **Middleware** [20], https://learn.microsoft.com/it-it/aspnet/core/fundamentals/middleware/?view=aspnetcore-8.0
- **Razor engine** [21], https://learn.microsoft.com/en-us/aspnet/core/mvc/views/razor?view=aspnetcore-8.0
- **LINQ (Language Integrated Query)** [22], https://learn.microsoft.com/it-it/dotnet/csharp/linq/
- **EF Core API Fluent** [23], https://learn.microsoft.com/en-us/ef/core/modeling/relationships
- **Tracking** [24], https://learn.microsoft.com/it-it/ef/core/change-tracking/
- **State Machine** [25], https://learn.microsoft.com/en-us/dotnet/framework/windows-workflow-foundation/state-machine-workflows
- **Layoyt** [26], https://learn.microsoft.com/it-it/aspnet/core/mvc/views/layout?view=aspnetcore-8.0
- **Tag Helper** [27], https://learn.microsoft.com/en-us/aspnet/core/mvc/views/tag-helpers/intro?view=aspnetcore-8.0

- **Datatable** [28], https://datatables.net/manual/index
- **Ajax** [29], https://api.jquery.com/category/ajax/
- **Leaflet** [30], https://leafletjs.com/reference.html
- **Geoapify** [31], https://www.geoapify.com/
- **Trello** [32], https://trello.com/it/guide
- **Sartorello Bruciatori** [33], https://sartorellobruciatori.it/

Definizioni

Definizioni:

- **Agile** (Fonte https://slack.com/intl/it-it/blog/collaboration/agile-project-management).
  L'agile project management è un approccio alla gestione dei progetti basato su **principi iterativi e incrementali**, con l'obiettivo di promuovere la **collaborazione**, la **flessibilità** e la **creazione continua di valore per il cliente**.
  Una delle caratteristiche principali dell'agile project management è **l'uso di sprint o sessioni brevi**, ossia la costruzione di un progetto in piccoli passi, dividendolo in più tappe e coinvolgendo il cliente frequentemente nelle prime fasi dello sviluppo, facilitando la flessibilità e l'adattamento ai cambiamenti.
  Inoltre, l'agile project management è caratterizzato da **un'attenzione particolare alla collaborazione e alla comunicazione efficace nei team**, incoraggiando la loro autogestione e motivazione.
- **Bloatware**, la tendenza dei nuovi software a diventare sempre più onerosi in termini di risorse hardware, senza che ci sia un miglioramento significativo della qualità del software per l'utente finale. (Fonte wikipedia [4])
- **Back end**, è la parte di un'applicazione o sito web che gestisce la logica di business, il database, l'autenticazione degli utenti e la comunicazione tra il server e il front-end. È responsabile dell'elaborazione delle richieste degli utenti, dell'interazione con il database e della gestione dei dati.
- **Front end**, è la parte di un'applicazione o sito web che interagisce direttamente con l'utente. Include l'interfaccia utente (UI) e la presentazione dei dati, ed è responsabile della visualizzazione delle informazioni e dell'interazione con l'utente tramite elementi come pulsanti, moduli e layout.
- **Backlog**, è una lista di compiti non ancora eseguiti, in arretrato o da portare a termine adesso. Uno strumento che permette di definire facilmente questo tipo di liste è **Trello** [32].
- **Object-Relational Mapper (ORM)**, in informatica è una tecnica di programmazione per convertire i dati tra un database relazionale e l'heap di un linguaggio di programmazione orientato agli oggetti. Ciò crea, in effetti, un database di oggetti virtuali che può essere utilizzato dall'interno del linguaggio di programmazione.
- **File compilati**, il C# essendo un linguaggio di programmazione compilato e non interpretato utilizza un compilatore che genera per ogni file di sorgete .cs (linguaggio ad alto livello) un corrispettivo file .dll, questo nuovo file viene scritto in un formato comprensibile direttamente dalla macchina (Ad esempio il risultato saranno istruzioni macchina oppure in un linguaggio intermedio).