ISTITUTO TECNOLOGICO SUPERIORE ACADEMY

PER LE TECNOLOGIE DELLA INFORMAZIONE E DELLA COMUNICAZIONE

Sede legale: Torino, Piazza Carlo Felice 18

DENOMINAZIONE CORSO : **Tecnico superiore per i metodi e le tecnologie per lo sviluppo di sistemi software - SOFTWARE DEVELOPER**

Sviluppo e Testing di una Web Application in ambiente .NET e Angular per la gestione di dispositivi di rilevamento dati

Candidato

RIVOIRA ALBERTO

Intervento realizzato da

SOMMARIO

1. INTRODUZIONE

Il presente elaborato si propone di descrivere l'attività di tirocinio curriculare svolta presso la società **Byte S.r.l.**, realtà operante nel settore della consulenza informatica, specializzata nello sviluppo di soluzioni software personalizzate per i vari clienti, con il principale utilizzo delle tecnologie **C#** e **Angular**.

Durante il periodo di stage, è stato possibile partecipare allo sviluppo di un progetto concreto, con l'obiettivo di approfondire l'utilizzo di tecnologie moderne per lo sviluppo di applicazioni web.

Il progetto in questione ha previsto la realizzazione di una web application, realizzata mediante il framework **Angular** per il front-end e il linguaggio **C#**, con l'utilizzo del framework **ASP.NET Core**, per la parte back-end. Tale applicazione si inserisce in un più ampio contesto aziendale volto alla digitalizzazione e all'automazione della gestione della raccolta di dati prevalentemente numerici.

Durante l'esperienza il lavoro è stato svolto insieme in un piccolo gruppo di persone, affiancati da un tutor esperto che si è sempre mostro presente e disponibile.

L'elaborato contiene una panoramica di tutto il percorso che ha coinvolto il progetto su cui si è svolto il tirocinio, delle tecnologie utilizzate e delle attività svolte, soffermandosi in particolare sulle fasi di analisi, progettazione e sviluppo del progetto assegnato.
Verranno inoltre evidenziate le principali problematiche affrontate e le soluzioni adottate nel corso dell'attività, con l'obiettivo di documentare il percorso formativo intrapreso.

Tutto il contenuto dell'elaborato verrà esposto in ordine cronologico e in ordine tematico, per cui sarà diviso in due principali capitoli, quello del progetto, il quale

conterrà in ordine cronologico i problemi affrontati e le attività svolte, separatamente divise tra Backend e Frontend.
Il secondo capitolo principale sarà quello degli approfondimenti, per cui anche qui i contenuti saranno in ordine cronologico e divisi tra Backend e Frontend, e conterrà le diverse spiegazioni teoriche su cui ci si è dovuti soffermare per risolvere vari problemi oppure per un approfondimento puramente teorico.

2. TECNOLOGIE USATE

Editor utilizzati:

Logo Microsoft Visual Studio

- Microsoft Visual Studio: ambiente di sviluppo integrato (IDE) sviluppato da Microsoft, ampiamente utilizzato per la creazione di applicazioni in C# e per progetti basati sul framework .NET. In questo progetto è stato impiegato per lo sviluppo della componente back-end, offrendo strumenti avanzati per il debugging, la risoluzione di conflitti, gestione dell'ambiente di sviluppo e l'esecuzione dei test automatici.

Logo Visual Studio Code

- Visual Studio Code: editor di codice sorgente leggero e multipiattaforma sviluppato da Microsoft, particolarmente adatto allo sviluppo web grazie al supporto esteso per linguaggi come JavaScript e TypeScript, e all'integrazione con numerose estensioni. In questo progetto è stato utilizzato per lo sviluppo della componente front-end in

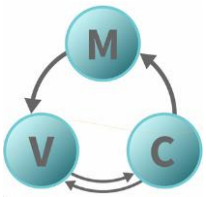
Angular, grazie alla sua flessibilità, alla gestione efficiente dei file di progetto e al supporto per il terminale integrato.

Backend:

Logo C#

- C#: un linguaggio di programmazione orientato agli oggetti sviluppato da Microsoft, ampiamente utilizzato per lo sviluppo di applicazioni desktop, web e cloud-based all'interno dell'ambiente .NET. Nel progetto in oggetto, C# è stato impiegato per la realizzazione dell'intera componente back-end, grazie alle sue funzionalità avanzate e al supporto nativo per il modello MVC.

Modello MVC



- Modello MVC: l'architettura software adottata per il progetto che suddivide l'applicazione in tre componenti principali: **Model** (gestione dei dati e della logica), **View** (interfaccia utente) e **Controller** (gestione delle interazioni e coordinamento tra Model e View), facilitando così la separazione delle responsabilità e la manutenibilità del codice.

Logo .Net-Core

- Framework .Net-Core 8: si tratta di una piattaforma ad alte prestazioni e open source, utilizzata per la realizzazione di applicazioni web, cloud e microservizi. In questo progetto, .NET Core 8 è stato impiegato per sviluppare la logica back-end, facendo uso del modello MVC e delle API REST.

Frontend:

Logo Angular

- Angular: framework open source sviluppato da Google per la

realizzazione di applicazioni web. Basato su TypeScript, Angular offre una struttura modulare e component-based, che favorisce la scalabilità e la manutenibilità del codice. In questo progetto è stato utilizzato per sviluppare l'intera interfaccia utente e gestire la comunicazione con il back-end tramite chiamate HTTP.

Logo PrimeNG

- PrimeNG: una libreria di componenti UI open source basata su Angular, che fornisce un'ampia raccolta di elementi grafici pronti all'uso, come tabelle, form, pulsanti e notifiche. In questo progetto è stata utilizzata per velocizzare lo sviluppo dell'interfaccia utente, garantendo al contempo coerenza stilistica e una buona esperienza d'uso.

Database:

Logo MySQL

- MySQL: sistema di gestione di basi di dati relazionali (RDBMS) open source tra i più diffusi, noto per l'affidabilità, le buone prestazioni e l'efficienza nella gestione di grandi quantità di dati. In questo progetto è stato utilizzato come database per la memorizzazione e la gestione delle informazioni, in combinazione con Entity Framework per l'accesso ai dati dal back-end.

Logo Docker

- Docker: piattaforma open source utilizzata per eseguire applicazioni in ambienti isolati chiamati container. In questo progetto è stato impiegato per avviare e ospitare il database MySQL in modo isolato e facilmente replicabile, semplificando la configurazione dell'ambiente di sviluppo e garantendo coerenza tra le diverse postazioni di lavoro.

Gestione del progetto:

Logo GitLab

- GitLab: piattaforma open source che integra funzionalità di versionamento del codice, gestione dei repository Git e strumenti per la collaborazione e l'automazione del ciclo di sviluppo. In questo progetto è stato utilizzato per il controllo versione del codice sorgente e per la gestione delle diverse branche di sviluppo, favorendo il lavoro collaborativo e la tracciabilità delle modifiche.

Logo Jira

- Jira: strumento di project management sviluppato da Atlassian, utilizzato principalmente per la pianificazione, il monitoraggio e la gestione di progetti software secondo metodologie agili. In questo progetto è stato impiegato per la gestione delle attività, l'assegnazione dei task e il tracciamento dell'avanzamento del lavoro, facilitando l'organizzazione del gruppo e la suddivisione delle fasi di sviluppo.

3. PROGETTO TRACKER

a. INTRODUZIONE AL PROGETTO

Il progetto su cui è stato svolto il tirocinio, d'ora in avanti denominato *Tracker*, consiste in una web application realizzata per la raccolta e la gestione di dati per conto del cliente, come ad esempio valori di temperatura e peso. L'applicazione era già in fase di sviluppo all'inizio del periodo di stage, ed erano state completate unicamente le funzionalità essenziali al corretto funzionamento dell'intero sistema.

Il progetto è stato sviluppato seguendo l'architettura MVC (*Model-View-Controller*): i componenti **Model** e **Controller** sono stati integrati nella parte back-end e sviluppati in linguaggio **C#**, mentre la componente **View** è stata gestita tramite il front-end, realizzato con il framework **Angular**.

Nel corso dello stage, oltre alle attività principali descritte nei capitoli successivi, sono state svolte con regolarità operazioni di verifica e correzione di possibili malfunzionamenti del sistema. Ogni anomalia rilevata è stata gestita attraverso la creazione di un'apposita **card su Jira** e lo sviluppo di un **branch dedicato su Git**. La maggior parte di questi interventi è consistita in attività di breve durata e risolte senza particolari complessità, pertanto non verranno trattate in dettaglio, trattandosi per lo più di operazioni specifiche e contestualmente difficili da inquadrare in modo esaustivo.

b. BACKEND

i. UNITTESTING

Come prima attività all'interno del tirocinio, è stato richiesto di dedicarsi alla scrittura di **unitTest** utilizzando il framework **xUnit**, comunemente adottato nello sviluppo in **.NET Core** per l'automazione dei test.

Questa fase iniziale è stata scelta con due obbiettivi principali: da un lato, apprendere il metodo corretto per sviluppare dei test in un contesto professionale, e dall'altro, favorire una comprensione approfondita del codice preesistente del progetto, senza intervenire direttamente su componenti critici o complessi.

L'attività di scrittura degli unitTest ha rappresentato un'opportunità formativa efficace per esplorare la struttura e l'architettura del software già sviluppato, familiarizzare con le convenzioni adottate dal gruppo di sviluppo, e comprendere la logica dietro tutte le componenti. Al contempo, è stato possibile consolidare le competenze relative all'utilizzo della Dependency Injection, al concetto di isolamento del codice fondamentali per un testing efficace.

Per questa prima attività non è stata necessaria una grande componente di analisi del lavoro, in quanto fosse solo da creare i test di tutti i service attualmente creati, dopo avere scritto una lista dei test che era necessario creare è stato deciso uno tra i più semplici ed è stato realizzato in coppia tra me e il mio collega dell'ITS con cui ho svolto questo periodo di stage, dopo di che sono stati divisi in modo equo i test da creare rimanenti, in base alla complessità dei service su cui si basavano.

Tutti i test sono stati sviluppati seguendo la seguente logica, è stata scritta una classe di test base che utilizza i tipi generici (così da rendere il codice più riutilizzabile e breve), la quale conteneva dei metodi di setup, come un metodo per generare il service necessario da testare, e un metodo che inseriva i dati dentro il database.

Dentro la classe base sono poi stati scritti tutti i test per i metodi base di un service CRUD (insert, update, delete…), con una variante che testava il corretto funzionamento

del metodo, e una seconda variante che ne verificava il previsto malfunzionamento nei casi in cui venivano lanciate delle eccezioni e verificate che fossero quelle previste.

Durante lo sviluppo dei test sono stati riscontrati diversi problemi, tra cui la modifica dei dati all'interno del principale database, cosa che non doveva avvenire, per cui si è stato deciso di creare un database con un numero randomico nel nome e che veniva quindi anche aggiunto dentro la stringa di connessione, questo è stato fatto avvenire per ogni singolo test, così che non si intralciassero a vicenda i vari test (per esempio test in cui si modificano o eliminano dati).

Mentre al termine di ogni test serviva che venisse quindi eliminato tale database, essendo che era a singolo utilizzo, è stata implementata l'interfaccia **IDisposable** che rende disponibile il metodo "Dispose" che viene automaticamente eseguito al termine del test, dentro al quale è stata scritta la logica per eliminare il database.

Un altro problema che è stato riscontrato era con alcuni test che prendevano i dati come parametri e quindi tramite dei metodi statici che li generavano, con cui c'è stata la possibilità di approfondire l'utilizzo e i rischi di proprietà o metodi statici.

L'ultima difficoltà rilevante che abbiamo dovuto risolvere è stata nei in cui il test doveva verificare che il codice si interrompesse nel punto e modo voluto, per cui le richieste mandate al database venivano bloccate prima di essere inviate, grazie al tutor che ha spiegato bene il funzionamento si è capito che l'elemento del databaseContext, il quale appunto esegue le richieste al database, quando una richiesta non va a buon fine la mantiene in memoria, e riprova ad eseguirla quando una seconda richiesta viene inviata, seguendo questo processo la prima ancora una volta fallisce facendo bloccare anche la seconda.

Dovendo risolvere questo problema si è dovuto andare a modificare la classe generica di base di tutti i service, per far sì che in tutte le interazioni con il database, venisse ripulito il tracker della memoria del context nei casi in cui veniva lanciata l'eccezione quando dei parametri non erano rispettati

Al termine dello sviluppo di tutte le classi di testing necessarie sono poi state controllate dal tutor e rese più efficaci dove possibile, corrette dove sbagliate così da avere un iniziale idea di come possa essere ottimizzato il codice.

ii. LOGICA RILEVAZIONI TEMPERATURA

Durante il percorso di tirocinio, l'attività iniziale assegnata ha riguardato l'aggiunta di una nuova entità all'interno dell'applicazione: il rilevamento della temperatura. Questa funzionalità ha richiesto un intervento su diversi strati dell'architettura back-end, dalla definizione del modello dati fino all'esposizione tramite API, garantendo il collegamento con il front-end.

La prima fase ha previsto la creazione del model RilevamentoTemperaturaEntity, che rappresenta il dato nel contesto del database. Parallelamente, è stato definito un DTO (RilevamentoTemperaturaDto) per il trasferimento dei dati verso il client, rispettando il principio di separazione tra logica di persistenza e logica esposta all'esterno.

Per gestire la conversione automatica tra entità e DTO è stato utilizzato **AutoMapper**, una libreria che semplifica la mappatura tra oggetti. All'interno del file AutomapperProfile.cs è stata specificata la corrispondenza tra RilevamentoTemperaturaEntity e RilevamentoTemperaturaDto, e viceversa. Ciò consente una trasformazione trasparente dei dati durante le operazioni di lettura e scrittura, migliorando la manutenibilità del codice.

Nel file ApplicationDbContext.cs, l'entità è stata aggiunta al metodo OnModelCreating, che permette di configurare i comportamenti e le relazioni tra le entità. In particolare, è stato utilizzato il metodo HasDiscriminator() per gestire l'ereditarietà tramite discriminatore, evitando la creazione di più tabelle separate per ciascuna tipologia di rilevamento.

Questa configurazione permette di salvare differenti tipologie di rilevamento in un'unica tabella, distinguendole tramite il campo TipologiaRilevamento, per cui è stata creata una classe di tipo Enum contenente i possibili valori. Il valore "Temperatura" rappresenta, in questo caso, la sotto-classe RilevamentoTemperaturaEntity.

Una volta definita la struttura, è stata creata e applicata una nuova **migration** tramite Entity Framework, per aggiornare lo schema del database in modo coerente con le modifiche apportate al modello. Questo processo ha generato il codice SQL necessario a creare (o modificare) le tabelle e le relazioni coinvolte.

Per quanto riguarda la logica applicativa, non è stato necessario creare un nuovo service da zero, in quanto era già presente una classe di base generica, la quale utilizzava i tipi generici, da cui i vari servizi ereditano funzionalità comuni come le operazioni CRUD. È stato quindi sufficiente creare una classe derivata, RilevamentoTemperaturaService, che eredita dalla base e sfrutta i metodi già implementati.

A questa è stato aggiunto un metodo specifico per recuperare l'ultima temperatura rilevata, ordinando i dati in base alla data di rilevamento e selezionando il valore più recente. Questo metodo rappresenta un'estensione della logica comune e mostra come la struttura ereditata consenta una facile espandibilità del codice, senza duplicare le funzionalità già esistenti.

Anche per il controller, è stato seguito un approccio basato sull'ereditarietà da una classe generica, già predisposta per la gestione delle operazioni CRUD standard tramite pattern RESTful. La creazione del controller RilevamentoTemperaturaController ha quindi previsto un'estensione minimale, ereditando i metodi comuni dalla classe base.

L'unica aggiunta ha riguardato un **endpoint personalizzato** per ottenere l'ultima temperatura rilevata. Questo è stato implementato come metodo HTTP GET, il quale richiama il metodo specifico del service e restituisce il valore più recente disponibile nel sistema. Tale approccio ha permesso di mantenere il controller snello, riutilizzando il più possibile la logica esistente e mantenendo la coerenza con gli altri componenti dell'applicazione.

iii. LOGICA CRONOLOGIA DISPOSITIVO-PROPRIETARIO

Un'ulteriore attività significativa ha riguardato l'introduzione di una funzionalità per la gestione dello storico delle assegnazioni tra dispositivi e proprietari. L'obiettivo era quello di mantenere una **cronologia dei cambi di proprietà** dei dispositivi di rilevamento, permettendo la tracciabilità delle assegnazioni nel tempo.

Prima di passare alla realizzazione della nuova sezione si è analizzato il problema, gli usi e le funzionalità necessarie allo sviluppo, fase in cui si è principalmente analizzato a livello di database come si volesse gestire la cronologia.

Inizialmente era stato deciso di implementare solo un campo con l'id del proprietario attuale nei dati del dispositivo, e una lista virtuale di id dei dispositivi posseduti nei dati del proprietario. Queste modifiche sarebbe inizialmente bastate dato che non era ancora stata proposta l'idea di una vera e propria cronologia, ma solo di visualizzare i dispositivi e il proprietario attuale.

Dopo un po' di ragionamento e del brainstorming di gruppo è stato deciso di effettivamente creare un'effettiva cronologia, e quindi serviva creare una nuova tabella intermedia nel database contenente id del proprietario vecchio, id del proprietario nuovo, id del dispositivo e la data del cambio di proprietario, in cui l'id del proprietario nuovo rimane nullo se il dispositivo non è più in uso.

Per realizzare tale funzionalità, è stato necessario intervenire su due entità principali già presenti nel sistema: **Dispositivo** e **Proprietario**. In particolare:

- Alla classe DispositivoEntity è stato aggiunto un campo per indicare il proprietario attuale.
- Alla classe ProprietarioEntity è stata aggiunta una lista di dispositivi posseduti, rappresentata come una relazione uno-a-molti.

Queste modifiche hanno permesso di rappresentare lo stato attuale delle assegnazioni nel database.

Per la gestione dello storico vero e proprio è stata introdotta una nuova entità CronologiaProprietarioDispositivoEntity, rappresentante ogni **evento di cambio proprietà**. Ogni record contiene le seguenti informazioni:

- L'ID del dispositivo coinvolto;
- Il vecchio proprietario;
- Il nuovo proprietario;
- La data del passaggio.

Tale entità è stata correttamente configurata nel metodo OnModelCreating del DbContext, e nell'automapperprofile per la conversione entità-dto, dove sono state

definite le relazioni tra le entità coinvolte e le regole di integrità referenziale. Inoltre, è stata generata una nuova migration per aggiornare la struttura del database e includere la tabella intermedia.

A supporto della nuova entità è stato definito un dto dedicato per la comunicazione con il front-end, seguito dalla creazione del corrispondente service, incaricato di gestire le operazioni di registrazione e consultazione dello storico. Il service ha previsto metodi specifici per l'inserimento di un nuovo passaggio di proprietà e per il recupero della cronologia relativa a un dato dispositivo.

Infine, è stato sviluppato il relativo controller, esposto tramite API REST, con endpoint per la registrazione del cambio di proprietario e per la visualizzazione dello storico. Questo ha permesso al front-end di interagire con il sistema in modo trasparente, visualizzando lo storico direttamente nell'interfaccia utente.

iv. RIFINITURA UNIT TEST CON CODE COVERAGE

Dopo una prima fase introduttiva dedicata alla scrittura di unit test per familiarizzare con il codice esistente, una seconda attività ha riguardato l'**analisi e integrazione di test mancanti**. Questo lavoro si è concentrato sull'individuazione di porzioni di codice non coperte da test automatici, con l'obiettivo di migliorarne l'affidabilità e la manutenibilità.

In progetti di medie o grandi dimensioni, è comune che alcune funzionalità non vengano testate in maniera esaustiva, soprattutto in fasi di sviluppo avanzate o in presenza di refactoring frequenti. Per affrontare questa criticità, è stata effettuata una revisione del codice sorgente e delle classi di test già presenti, al fine di:

- individuare metodi pubblici privi di test
- rilevare casistiche non coperte (es. gestione degli errori, condizioni limite)
- verificare sul totale del codice quanto fosse coperto dai test in percentuale

Il lavoro ha previsto l'aggiunta di test unitari focalizzati su rami logici trascurati, come quelli legati al lancio di eccezioni, alla validazione dei parametri e alla corretta gestione dei dati in condizioni particolari.

Per supportare e validare questo processo è stato introdotto il concetto di **test coverage** (copertura del codice), una metrica che indica quale percentuale del codice applicativo viene effettivamente eseguita durante l'esecuzione dei test. Sebbene un'elevata copertura non garantisce da sola l'assenza di bug, rappresenta un buon indicatore della qualità e della profondità dei test scritti.

La copertura è stata misurata in termini di:

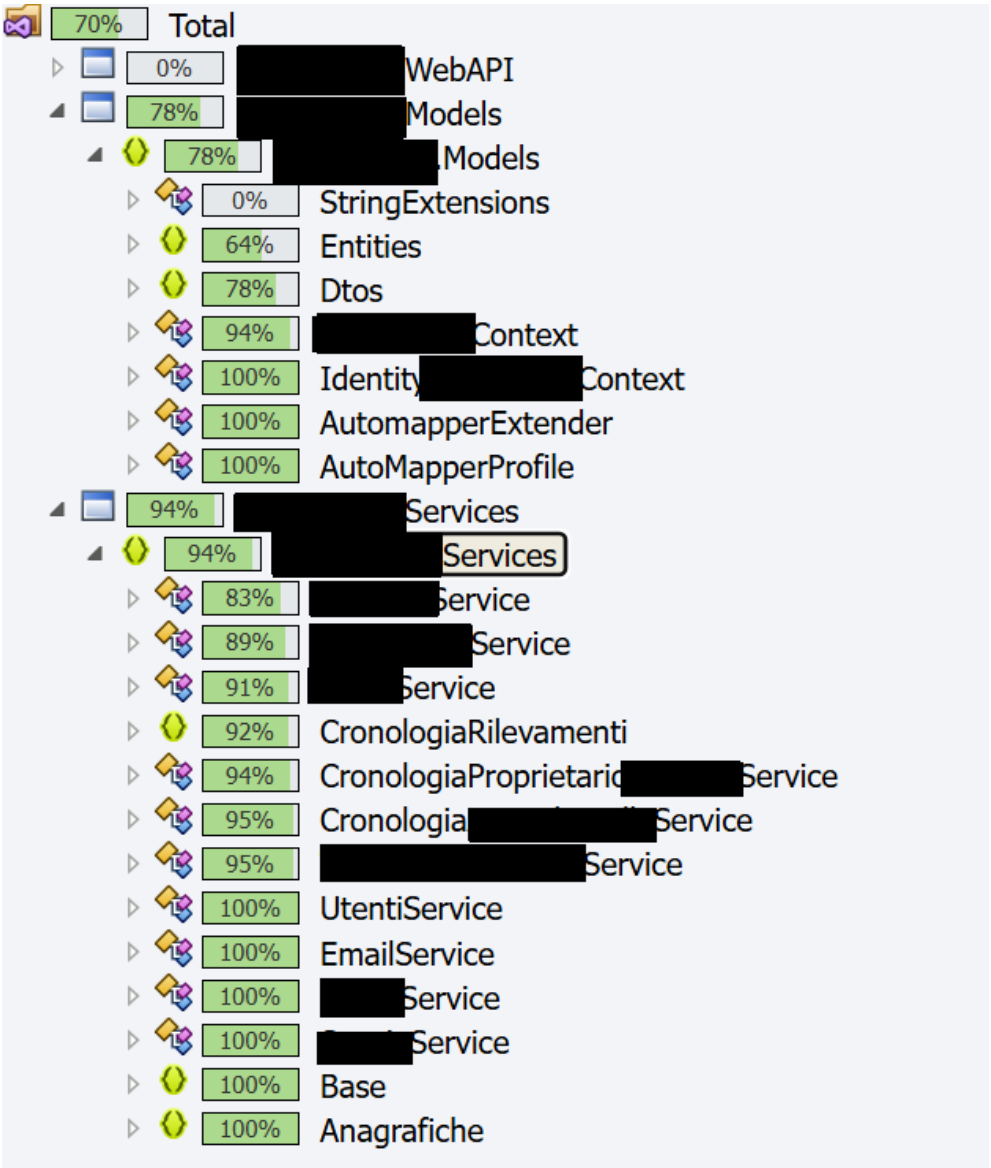
- **Line coverage** – percentuale di righe di codice effettivamente eseguite
- **Branch coverage** – percentuale di rami condizionali (es. if/else) verificati durante i test
- **Method coverage** – percentuale di metodi chiamati almeno una volta

Per monitorare in modo visivo la copertura del codice è stata utilizzata l'estensione **Fine Code Coverage** integrata in Visual Studio. Questa estensione consente di evidenziare direttamente nel codice le righe coperte (in verde), parzialmente coperte (in giallo) o

non coperte (in rosso), offrendo un'interfaccia chiara e immediata per valutare l'efficacia dei test.

L'adozione di questo strumento ha permesso di intervenire in maniera mirata, migliorando la copertura in classi critiche o particolarmente soggette a modifiche. Inoltre, ha facilitato l'individuazione di blocchi di codice non raggiunti da alcun test, portando alla scrittura di nuovi casi specifici.

Tramite l'utilizzo di questa estensione si poteva anche ottenere un file esterno, il quale mostrava sia in percentuale, sia nello specifico, quanti e quali porzioni di codice fossero mancanti di testing e ci si è potuto lavorare (sotto l'immagine del report)



Report Code Coverage

Come si è potuto osservare tramite il report c'erano delle parti totalmente non toccate dai test attualmente esistenti, ed altre per cui i test già esistevano, ma di cui mancavano alcuni metodi o sezioni dei metodi, come verifica delle eccezioni o verifica di tutti i casi di codizioni if/else.

In particolare si è dovuto lavorare sui test per altri service nuovi, e alcuni metodi comuni a tutte le classi che inizialmente non erano stati ritenuti necessari di testing.

Un test in particolare che ha creato problemi è stato il test per il metodo che recuperava l'ultima rilevazione di un certo dispositivo (menzionato in un capitolo precedente), in quanto bisognava anche verificare che il dato fosse effettivamente l'ultimo, per cui si è ricorsi all'utilizzo della libreria MOQ (spiegazione nella sezione approfondimenti) per simulare un database con dei dati preimpostati inseriti dal test.

Questa seconda fase di lavoro sui test ha rappresentato un'importante opportunità per approfondire concetti legati alla **qualità del software**, al controllo delle regressioni e all'importanza di una **test coverage consapevole**, che non si limiti a massimizzare la percentuale, ma miri a garantire una reale protezione contro comportamenti imprevisti.

c. FRONTEND

i. PROGETTO PROVA

Nel corso del tirocinio, prima di essere coinvolti nello sviluppo diretto dell'applicazione principale, è stato assegnato un progetto di prova volto a familiarizzare con l'utilizzo del framework **Angular** e con i principi fondamentali dello sviluppo front-end. L'esercitazione ha previsto la realizzazione di una simulazione di carrello della spesa, in cui fosse possibile aggiungere, rimuovere e modificare articoli, con aggiornamento dinamico delle quantità e del totale del prezzo.

Il progetto ha permesso di approfondire concetti chiave come la **gestione dello stato del componente**, l'uso del **data binding** (sia unidirezionale che bidirezionale), l'interazione con eventi utente, nonché la manipolazione dinamica delle liste e l'utilizzo di servizi per la gestione centralizzata dei dati. Attraverso questa attività, è stato possibile acquisire maggiore domestichezza con la struttura modulare di Angular e con le pratiche comuni nella realizzazione di interfacce utente reattive e funzionali.

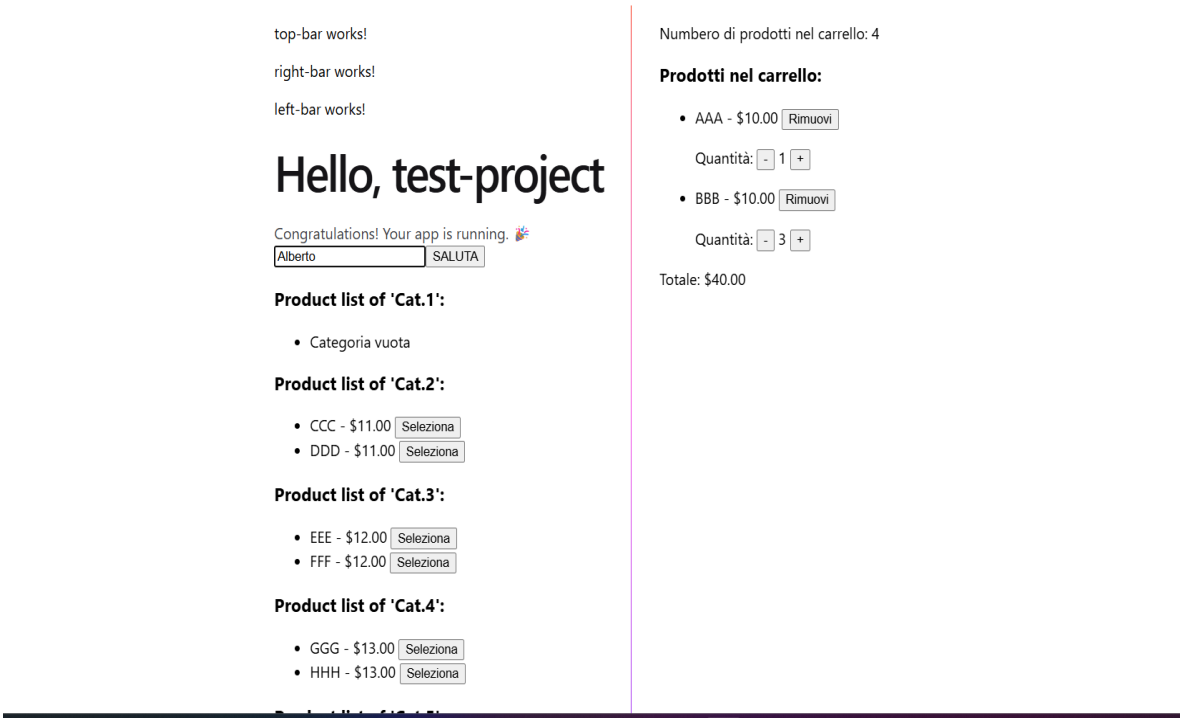
Durante lo sviluppo di questo piccolo progetto si sono svolte delle attività graduali e complementari l'una con l'altra, inizialmente si è solo sviluppato un singolo componente di angular il quale conteneva solo del testo per prendere mano con tutte le componenti di base e i funzionamenti iniziali. Come secondo passo è stato fatto comparire un alert come pop-up che saluta il nome dato in input, così da imparare il data binding con l'esecuzione di un metodo legato a questo dato che si aggiorna dinamicamente.

Dopo aver visto i punti cruciali dello sviluppo in angular, si è iniziato a sviluppare l'effettivo carrello della spesa che si aggiornasse in modo dinamico, iniziando dall'avere dei componenti secondari creati dentro il primo, che avessero dei dati legati a parti esterne del componente in cui sono dichiarati cosi da vederne i comportamenti, e capire come interagirci nel modo appropriato; aggiungendo poi altri componenti dello stesso tipo, tramite un ciclo for che ne aggiungeva tanti quanti il numero di categorie inserite in un array, che conteneva i dati da mostrare (i quali erano inseriti manualmente non essendo una web-app completa).

Il principale problema che è stato affrontato era con la modifica di dati in un componente quando i dati di un secondo componente erano modificati, i prodotti disponibili con quelli del carrello, il quale subito era stato parzialmente risolto con le proprietà get/set e altri metodi di supporto, ma anche con questi non funzionava sempre nel modo corretto e in particolare era stato riscontrato un bug nell'utilizzo di queste proprietà dovuto ad una nuova versione di angular.

Il bug in questione faceva avvenire quattro volte le chiamate delle proprietà get/set invece che una come previsto; quindi, si è dovuta trovare un'altra soluzione, che grazie ad un suggerimento del tutor, è stata trovata tramite la ricerca e studio delle relazioni signal/computed e subject/subscriber

Infine si sono aggiunte le interazioni necessarie per far sì che le varie sezioni diverse dei "prodotti disponibili" e del "carrello" interagissero nel modo giusto tra di loro, il che comprendeva l'aggiornamento del contenuto, delle quantità e del prezzo totale degli elementi nel carrello, la rimozione di tali prodotti dalla lista dei disponibili, e il ritorno dei prodotti nella categoria corretta dei prodotti disponibili se rimossi dal carrello, per avere come risultato finale l'immagine sottostante.



ii. VISTA DEI RILEVAMENTI TEMPERATURA

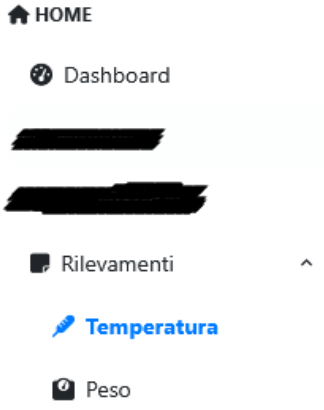
La prima attività svolta sul front-end del progetto, è stata l'aggiunta della view dei rilevamenti della temperatura, la quale consisteva in una pagina del sito in cui si potessero osservare le temperature rilevate da dai dispositivi di rilevazione e quando siano state rilevate, il tutto con i possibili ordinamenti (cronologico, alfabetico, crescente o decrescente) e con la possibilità di filtrare in modo generico tramite testo, con un arco temporale, o un intervallo di temperatura.

Una volta creata questa sezione doveva essere collegata anche al menu delle sezioni accessibili, tramite il router del componente del menu, con il giusto reindirizzamento verso il componente selezionato.

Per questa nuova funzionalità non è servito fare tanta analisi sullo sviluppo in quanto serviva solo a mostrare dei dati, mentre invece di è fatto un ragionamento di gruppo sul dove inserirla nella struttura già esistente del sito. Infine, si è deciso di inserire una sezione del menu per navigare nel sito, la quale doveva essere un menu a tendina chiamato rilevazioni, che, se esteso, mostrava tutti i link alle pagine delle varie rilevazioni in cui è stato inserito quello per le temperature, e che in futuro conterrà anche i link agli altri tipi di rilevazioni.

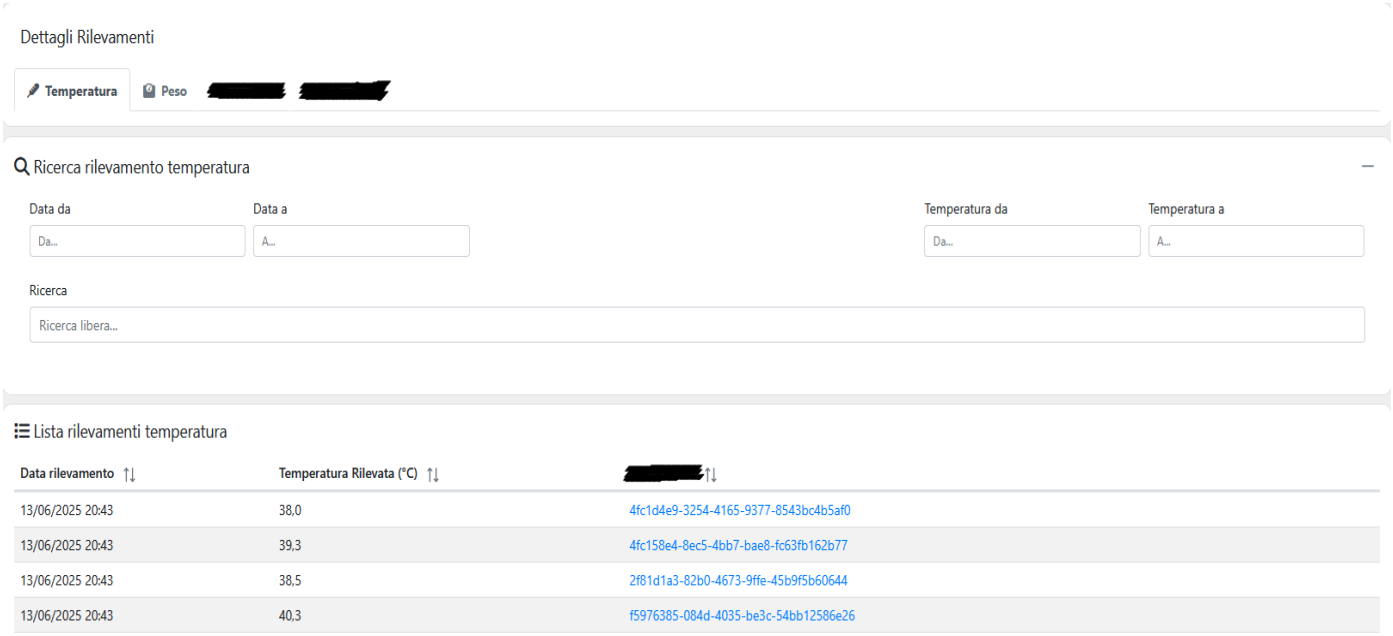
Nello sviluppo di questo componente si è potuta esplorare bene anche la struttura di tutto un progetto in angular, che per questa fase di sviluppo il tutto era strutturato con un componente per il menù a parte, e un componente esterno per le rilevazioni con dentro il componente temperatura, così che, se si volessero aggiungere altre rilevazioni di tipologia diversa in futuro si sarebbe potuto fare facilmente, e doveva quindi essere messo un collegamento dal menu alla pagina delle rilevazioni.

L'unico problema riscontrato che è importante menzionare, è stato nel creare il collegamento dal menu (nella foto a fianco) alla pagina delle rilevazioni (nella foto a fine capitolo), perché non riuscivamo a fargli prendere i dati di un componente totalmente esterno a quelli interni e



Menu Sito

viceversa, dopo varie ricerche e tentativi non si era trovato nulla che ci permettesse di risolvere ciò, e con l'aiuto del tutor si è potuto comprendere meglio il funzionamento del *routing* di angular, degli *import* e *declarations*, con cui si è riuscito a svolgere lo sviluppo di questo primo componente del front-end.



Pagina rilevamenti temperatura

iii. VISTA CRONOLOGIA DISPOSITIVO-PROPRIETARIO

Una volta completata l'implementazione della logica back-end relativa allo storico delle assegnazioni tra dispositivi e proprietari, si è proceduto con l'integrazione della nuova funzionalità anche all'interno dell'interfaccia utente, sviluppata in Angular.

L'obiettivo principale era offrire una navigazione chiara e intuitiva della cronologia, sia a partire dalla vista dei dettagli di un proprietario, sia da quella di un dispositivo, con rappresentazioni visive che facilitassero la consultazione dei dati storici.

In questa fase dello sviluppo non è stato necessario fare una grossa parte di analisi, in quanto è stata sviluppata dopo la logica del backend, e quindi esisteva già un'idea di come implementarlo a livello di grafica. L'unica parte di analisi è stata fatta a metà dello sviluppo in cui si è deciso di voler aggiungere una timeline, per cui si è ragionato come implementarla e dove posizionarla, infine è stato deciso di aggiungerla sotto la tabella in posizione orizzontale.

All'interno della pagina dei dettagli del proprietario, è stato aggiunto un collegamento diretto alla sua cronologia di dispositivi. Questo collegamento reindirizza a una nuova pagina dedicata, che presenta due sezioni principali:

- Una **tabella**, che elenca in modo dettagliato tutti i dispositivi posseduti nel tempo dal proprietario selezionato, indicando le date di inizio e fine assegnazione, i codici identificativi dei dispositivi e il nome del nuovo proprietario;
- Una **timeline orizzontale**, che visualizza in ordine cronologico i vari passaggi di dispositivi al proprietario. Ogni evento della timeline mostra visivamente il momento dell'assegnazione, evidenziando l'evoluzione storica nel tempo.

La combinazione delle due sezioni consente all'utente di accedere rapidamente sia a una vista più completa con tutti i dati, che a una rappresentazione più grafica dello storico.

In maniera speculare, è stato introdotto un **collegamento analogo** anche all'interno della pagina dei dettagli del dispositivo, che rimanda a una vista dedicata alla cronologia dei suoi proprietari passati. Anche in questo caso, la pagina contiene:

- Una **tabella** con la lista dei proprietari precedenti e relativi periodi di assegnazione;
- Una **timeline** che mostra, in modo grafico, i cambi di proprietà del dispositivo nel tempo, con indicazione del passaggio da un proprietario all'altro.

Per la realizzazione di queste interfacce sono stati impiegati componenti forniti da **PrimeNG**, in particolare per la gestione delle tabelle (p-table) e per la visualizzazione della sequenza temporale (p-timeline). L'integrazione con le API è avvenuta tramite i

consueti servizi Angular, utilizzando chiamate HTTP per ottenere i dati storici dal back-end in base all'ID del proprietario o del dispositivo selezionato.

Durante lo sviluppo di queste nuove pagine sono stati riscontrati problemi in particolare con la timeline, in quanto era un componente ancora non utilizzato, ed essendo fornito in modo preimpostato tramite PrimeNG non possedeva tante opzioni di modifica.

Utilizzando alcune delle poche impostazioni che aveva il componente si è infine riusciti a adattarlo allo stile della pagina, e all'utilizzo che ne si voleva fare.

d. BRANCH MANAGEMENT

i. GIT LAB

Durante l'intera durata del tirocinio, il sistema di **versionamento del codice** è stato gestito tramite **GitLab**, una piattaforma DevOps basata su Git che permette di gestire repository, monitorare modifiche e facilitare il lavoro in gruppo attraverso strumenti di collaborazione e automazione.

GitLab è stato impiegato per salvare e condividere il codice sorgente dei vari moduli dell'applicazione, garantendo tracciabilità e sicurezza. Ogni nuova funzionalità è stata sviluppata in un ramo (*branch*) dedicato, seguendo una convenzione condivisa per i nomi (*nomeProgetto-N°Branch-nomeBranch*), per poi essere integrata nel ramo principale (*main*) tramite le **merge request**, previo rebase del branch sul main così da risolvere i conflitti in locale da chi ha sviluppato il branch e ne conosce il contenuto, con successiva revisione del codice da parte del tutor.

Grazie a GitLab è stato inoltre possibile:

- Mantenere uno **storico dettagliato** delle modifiche apportate,
- Effettuare **code review** tramite i commenti sulle merge request,

Questa esperienza ha permesso di acquisire maggiore familiarità con i concetti fondamentali del versionamento distribuito e con le buone pratiche di sviluppo collaborativo, come il commit frequente e descrittivo, la gestione dei conflitti in

particolare, avendo dovuto fare anche più di una volta il rebase del branch di lavoro, sul main dentro il quale erano state accettate delle merge request. Ciò richiedeva di verificare che non ci fossero dei conflitti con il codice nuovo, con cui spesso si avevano delle situazioni con dei conflitti su cui serviva lavorare in più di una persona, o anche dei finti conflitti generati da errori del programma.

ii. JIRA

Per la pianificazione e la gestione delle attività di sviluppo è stato utilizzato **Jira**, una piattaforma di project management orientata alla metodologia Agile. In particolare, Jira è stato impiegato per:

- La **definizione dei task** da assegnare ai vari membri del gruppo,
- Il monitoraggio dell'avanzamento delle attività,
- L'organizzazione del lavoro attraverso la **board Kanban**, con la gestione visuale delle card.

All'interno della board, ogni attività era rappresentata da una **card**, etichettata in base alla tipologia (es. bug, task, story) e spostata lungo le varie fasi (To Do, In Progress, In Review, Done) in base allo stato di avanzamento.

Durante il tirocinio si è potuto avere un primo approccio guidato a Jira, e la corretta creazione e gestione di card, in quanto dopo i primi utilizzi, ognuno si doveva creare e gestire le proprie card in base a come riteneva più giusto. Questo ha svolto un ruolo importante per apprendere come avviene la **gestione strutturata del lavoro in gruppo**, la priorizzazione delle attività, la stima delle tempistiche e l'importanza della documentazione associata ad ogni ticket.

4. APPROFONDIMENTI

a. APPROFONDIMENTI BACKEND

i. UNITTESTING

Uno *unitTest* è un blocco di codice che verifica l'accuratezza di un blocco di codice applicativo più piccolo e isolato, in genere una funzione o un metodo. Lo *unitTest* è progettato per verificare che il blocco di codice funzioni come previsto, secondo la logica teorica seguita dallo sviluppatore. Esso è in grado di interagire con il blocco di codice solo tramite gli input e i relativi output di risposta, decisi in base al ruolo inteso per il test.

Tali test sono progettati per essere eseguiti rapidamente e frequentemente durante il processo di sviluppo, contribuendo a individuare tempestivamente eventuali errori logici, regressioni o anomalie nel funzionamento delle componenti testate.

Gli unitTest possono essere differenziati in due maggiori categorie, quella più base a cui è attribuito il tag **[Fact]** è più semplice e immediata in quanto non prende parametri esterni e viene eseguito solo una volta. Mentre la seconda categoria, avente il tag **[Theory]** viene usata quando lo stesso test necessita di essere eseguito più volte con dei set di dati diversi.

Nella categoria theory i dati per il test possono anche essere passati in più modi:

- **[InlineData]**
  Consente di specificare direttamente i valori da utilizzare come parametri nel test. È il metodo più semplice e leggibile per casi statici e di piccole dimensioni.
- **[MemberData]**
  Permette di fornire i dati da una proprietà, un metodo o un campo statico. È utile per casi più complessi, o quando si desidera riutilizzare i dati tra più test.
- **[ClassData]**
  Utilizza una classe esterna che implementa l'interfaccia IEnumerable<object []>.

Questo approccio è indicato quando i dati da fornire sono particolarmente numerosi o generati dinamicamente.

ii. MOQ

Nello sviluppo di test unitari, un aspetto fondamentale è la possibilità di **isolare l'unità di codice da testare**, evitando che interazioni con dipendenze esterne (come database o altri servizi) influenzino l'esecuzione o il risultato del test. Per ottenere questo comportamento, viene comunemente adottato il concetto di **mocking**, ovvero la sostituzione delle dipendenze reali con versioni simulate, controllabili e prevedibili.

Durante le attività di test, è stata utilizzata in modo estensivo la libreria **Moq**, una delle più diffuse librerie per il mocking nel contesto .NET. Moq consente di creare oggetti "finti" (mock) che implementano interfacce o classi virtuali, permettendo di definire comportamenti personalizzati e di verificare le chiamate effettuate al loro interno.

Uno degli scenari principali in cui è stato impiegato Moq riguarda la **simulazione dei repository e del database**. Poiché un test unitario non dovrebbe accedere direttamente a un database reale (per evitare effetti collaterali e dipendenze esterne), il context viene solitamente sostituito, e la sua implementazione reale viene sostituita con una versione mock durante il test.

Un altro ambito in cui Moq si è dimostrato utile è nella **simulazione di altri servizi interni**, ad esempio per testare un service A che dipende da un service B. In questo caso, l'obiettivo è verificare il comportamento del service A, senza eseguire realmente la logica implementata nel service B.

Esempio:

```
var mockScheduler = new Mock<IScheduler>();
var mockSchedulerFactory = new Mock<ISchedulerFactory>();
mockSchedulerFactory.Setup(f =>
    f.GetScheduler(It.IsAny<CancellationToken>())).ReturnsAsync(mockScheduler.Object);
```

Moq permette anche di verificare che un certo metodo sia stato chiamato, con o senza determinati parametri. Questa funzionalità è utile per assicurarsi che l'unità sotto test si comporti come previsto, anche dal punto di vista delle interazioni.

Per esempio, serve verificare il funzionamento di un metodo A, il quale all'interno contiene una chiamata al metodo B, che a sua volta è già stato testato in un'altra sezione e non necessita verifiche, e di essere quindi eseguito avendo la possibilità che causi errori a sua volta, influenzando il metodo A.

In questo caso, quindi, viene fatto un Moq del service B, per far sì che il metodo chiamato da quel service, esegua una versione finta del metodo che nella pratica non fa nulla, ma permette di verificare che il metodo A arrivi a quel punto del codice senza errori.

Esempio:

```
MockEmailService.Verify(
    _ => _.SendConfirmRegistrationEmail(It.Is<IdentityUser>(u => u.UserName == dto.Username),
        confToken.EmailConfirmationToken),
    Times.Once);
```

Con questa istruzione, si verifica che il metodo SendConfirmRegistrationEmail sia stato chiamato esattamente una volta durante il test.

L'adozione di Moq ha portato diversi vantaggi nel processo di testing:

- **Isolamento completo** delle unità testate, senza dipendenze da database o servizi reali.
- **Controllo totale** sul comportamento delle dipendenze simulate.
- **Maggiore affidabilità e velocità** dei test.
- Possibilità di **verificare le interazioni** tra componenti, non solo i risultati finali.

iii. CICLO DI VITA DEI SERVIZI

Nel contesto dello sviluppo back-end con .NET Core, uno degli aspetti fondamentali della configurazione dell'applicazione riguarda la **gestione del ciclo di vita dei servizi**, ovvero la modalità con cui le dipendenze vengono istanziate e gestite nel tempo. Questo meccanismo è parte integrante del **dependency injection container**, che consente di registrare e risolvere i servizi nel corso dell'esecuzione.

Durante l'esperienza di tirocinio, la comprensione e il corretto utilizzo del ciclo di vita dei servizi si sono rivelati cruciali nella progettazione dei componenti dell'applicazione, sia per motivi di prestazioni che per evitare errori legati alla gestione dello stato.

- **Tipologie di ciclo di vita**

In .NET i servizi possono essere registrati nel container di dependency injection secondo tre principali modalità:

1. Transient

I servizi registrati come *Transient* vengono creati **ogni volta che vengono richiesti**. Ciò significa che una nuova istanza viene generata per ogni utilizzo, anche all'interno della stessa richiesta HTTP.

Vantaggi:

- Utile per oggetti leggeri e stateless.
- Nessuna condivisione non intenzionale di dati.

Svantaggi:

- Maggiore consumo di memoria in presenza di richieste ripetute.

2. Scoped

I servizi *Scoped* vengono istanziati **una sola volta per ogni richiesta HTTP**. All'interno della stessa richiesta, ogni dipendenza riceverà la stessa istanza del servizio.

Vantaggi:

- Ideale per operazioni che richiedono consistenza nello stato durante l'intera richiesta, come l'accesso al database.

Svantaggi:

- Non è possibile utilizzarlo in contesti singleton (come nei middleware globali), pena errori di scope.

3. Singleton

I servizi *Singleton* vengono istanziati **una sola volta per tutta la durata dell'applicazione**. La stessa istanza viene condivisa in tutte le richieste e componenti.

Vantaggi:

- Prestazioni elevate in quanto viene riutilizzata la stessa istanza.
- Ideale per servizi che mantengono dati globali o configurazioni.

Svantaggi:

- Se il servizio contiene stato modificabile, è necessario gestire correttamente la concorrenza.

- **Considerazioni pratiche**

Nel progetto sviluppato durante il tirocinio, si è fatto uso principalmente di **Scoped** per i servizi che interagiscono con il **database context** (DbContext), in quanto questa modalità garantisce che le operazioni sul database siano consistenti nell'ambito della stessa richiesta.

I servizi stateless, come quelli contenenti logiche di trasformazione o validazione, sono stati registrati come **Transient**, mentre configurazioni globali o manager condivisi sono stati registrati come **Singleton**.

5. CONCLUSIONI

L’esperienza di tirocinio presso l’azienda ByteSrl ha rappresentato un’importante occasione di crescita professionale e tecnica, permettendo un primo confronto diretto con le metodologie, le tecnologie e gli strumenti propri dello sviluppo software in contesto aziendale.

Durante il percorso, è stato possibile approfondire l’intero ciclo di vita di un progetto web, partecipando sia alla manutenzione e al testing di codice già esistente, sia allo sviluppo di nuove funzionalità, seguendo buone pratiche come l’utilizzo del pattern MVC, la scrittura di test automatici con xUnit, l’uso della libreria Moq per il mocking dei servizi, e l’applicazione coerente della dependency injection con corretta gestione del ciclo di vita dei servizi.

Inoltre, è stato dato ampio spazio all’apprendimento degli strumenti di lavoro collaborativo più diffusi nel settore, come GitLab per il versionamento e Jira per l’organizzazione delle attività, comprendendo l’importanza della revisione del codice e della tracciabilità delle modifiche.

L'approccio graduale all'inserimento nei flussi di lavoro — a partire dall'esplorazione del progetto attraverso gli unit test, fino allo sviluppo completo di nuove entità e funzionalità — ha permesso di acquisire una visione concreta e strutturata dello sviluppo software moderno.

In conclusione, il tirocinio non solo ha rafforzato le competenze tecniche già acquisite nel percorso di studi, ma ha anche introdotto nuove conoscenze fondamentali per l’inserimento nel mondo del lavoro, fornendo una solida base su cui costruire il proprio futuro professionale nel settore dell’ingegneria informatica.

6. SITOGRAFIA

- Capitolo “UnitTesting”: https://aws.amazon.com/it/what-is/unit-testing/
- Capitolo “MOQ”: https://github.com/devlooped/moq
- Capitolo “Ciclo di vita dei servizi” https://learn.microsoft.com/it-it/aspnet/core/blazor/components/lifecycle?view=aspnetcore-9.0