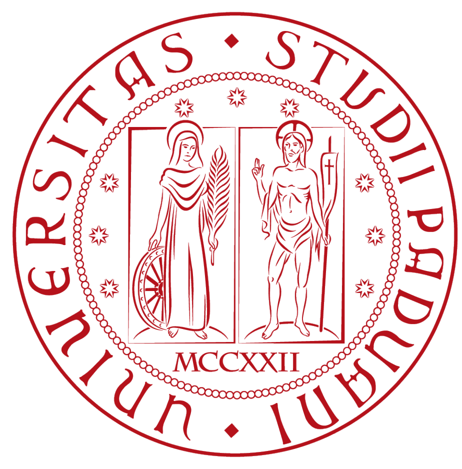
Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea in Informatica



Progettazione e sviluppo di un'applicazione moderna con il framework Blazor dedicata alla gestione dei condifi

Tesi di laurea

Relatrice

Prof.ssa Ombretta Gaggi

Laureanda

Angela Arena

1226299

Anno Accademico 2021-2022

Angela Arena: *Progettazione e sviluppo di un'applicazione moderna con il framework Blazor dedicata alla gestione dei condifi,* Tesi di laurea, © Dicembre 2022.

Stay hungry, stay foolish

— Steve Jobs

Dedico questa tesi, che racchiude un percorso importante e fondamentale della mia vita, a **Me** e alla **mia Nonna Carla**, che purtroppo in questo momento non può festeggiare personalmente con me e la mia famiglia, ma sono sicura che mi stia guardando dal Cielo, fiera ed orgogliosa, insieme ai mie nonni.

Sommario

Il presente documento descrive il lavoro che è stato svolto durante il periodo di stage, della durata di 304 ore, dalla laureanda Angela Arena presso l'azienda Galileo Network S.p.A.
L'obiettivo era quello di sviluppare un'applicazione web moderna con Blazor: in particolare realizzare ed implementare delle pagine web contenenti informazioni personali riguardo ad un'azienda-cliente. Più precisamente queste informazioni sono dati anagrafici dell'azienda-cliente e grafici esplicativi dei costi suddivisi per tipo di finanziamento.

Ringraziamenti

Vorrei ringraziare la Professoressa Ombretta Gaggi per i consigli, la disponibilità e il supporto fornitomi durante il periodo di stage e durante l'intera stesura della tesi.

Vorrei ringraziare l'azienda Galileo Network S.p.A per la possibilità che mi è stata data e, in particolare il mio tutor, per il costante supporto e la professionalità con cui è stato svolto il progetto.

Ringrazio immensamente la mia famiglia che mi ha sempre sostenuto, soprattutto nel momento del bisogno, e che mi ha motivato sempre a continuare, credendo in me. Grazie per avermi dato la possibilità di intraprendere questa strada, senza mai mettermi fretta e senza mai pretendere troppo. Sicuramente, senza il vostro aiuto, non sarei mai potuta diventare.. anche io.. Dottoressa.

Ringrazio Lorenzo, il mio fidanzato, per avermi dato forza nei momenti più difficili e bui, per avermi spronato a dare sempre il meglio di me, per aver gioito insieme al superamento di un esame e per avermi fatto sentire all'altezza anche quando un esame non andava come sperato. Senza il tuo aiuto non sarei arrivata fino a qui.

Ringrazio i miei amici di sempre, Anna, Linda e Niccolò, perchè da 16 anni ci sosteniamo a vicenda, dalla prima verifica alla scuola elementare all'ultimo esame universitario fatto. Siete stati sempre presenti per aiutarmi durante il mio percorso.

Ringrazio tutti i miei amici che hanno sempre creduto in me e che mi hanno sostenuta durante la mia carriera universitaria, motivandomi a dare sempre il meglio.

Padova, 16 Dicembre 2022 *Con affetto, Angela*

Indice

1 Introduzione **1**
1.1 Presentazione dell'azienda . . . 1
1.1.1 Confidi . . . 1
1.1.2 Match! . . . 1
1.2 Presentazione dello stage . . . 2
1.3 L'idea . . . 2
1.3.1 Obiettivo . . . 2
1.4 Organizzazione del testo . . . 2

2 Tecnologie **5**
2.1 Blazor . . . 5
2.1.1 ASP.NET Core . . . 5
2.1.2 Blazor Server . . . 6
2.1.3 Blazor WebAssembly . . . 6
2.2 SQL Server Management Studio . . . 7
2.3 Visual Studio 2022 . . . 7
2.4 GitLab . . . 8
2.5 Microsoft Teams . . . 9
2.6 Pulse Secure . . . 9

3 Analisi dei requisiti **11**
3.1 Casi d'uso . . . 11
3.1.1 Attori . . . 11
3.1.2 Classificazione . . . 11
3.1.3 Descrizione . . . 11
3.2 Elenco Casi d'Uso . . . 12
3.3 Tracciamento dei requisiti . . . 25
3.3.1 Classificazione . . . 25
3.3.2 Elenco Requisiti . . . 25

4 Progettazione **29**
4.1 Design Pattern utilizzati . . . 29
4.1.1 Model-View-Controller . . . 29
4.1.2 Dependency Injection . . . 30
4.1.3 AddScope in Blazor . . . 31
4.2 Struttura della soluzione Blazor . . . 32
4.3 Developer Express . . . 33

5 Sviluppo **35**

5.1 Struttura di una pagina Blazor 35
5.1.1 Componente Razor 35
5.1.2 Classe C# 35
5.1.3 Applicazione CSS 36
5.2 Component Loader 36
5.3 Implementazione dei widget 37
5.3.1 Widget Container 38
5.3.2 Donut Chart 39
5.3.3 Point Series Chart 41
5.3.4 Elenco Importi Totali 43
5.3.5 Popup Scelta Widget 44
5.4 Sviluppo di un form 46
5.5 Sviluppo di una griglia 47
5.5.1 Griglia con le operazioni di new, change e delete 49
5.5.2 Griglia con raggruppamenti 50
5.6 Internazionalizzazione 51
5.6.1 Utilizzo di IStreamLocalizer<T> 51
5.6.2 Accesso diretto al file di risorse 52

6 Verifica e validazione **54**
6.1 Analisi statica 54
6.2 Analisi dinamica 54
6.3 Test di accessibilità 54
6.4 Test dei messaggi d'errore di un form 55

7 Conclusioni **58**
7.1 Raggiungimento degli obiettivi 58
7.2 Requisiti soddisfatti 58
7.2.1 Requisiti funzionali soddisfatti 58
7.2.2 Requisiti qualitativi soddisfatti 60
7.2.3 Requisiti di vincolo soddisfatti 60
7.3 Valutazione personale 60

Glossario **62**

Elenco delle figure

2.1 Logo di Blazor . . . 5
2.2 Compilazione di un'app Blazor WebAssembly . . . 7
2.3 Logo di SQL Server Manager Studio . . . 7
2.4 Logo di Visual Studio 2022 . . . 8
2.5 Logo di GitLab . . . 8
2.6 Logo di Microsoft Teams . . . 9
2.7 Logo di Pulse Secure . . . 9

3.1 Use Case UC1 - Visualizzazione Donut Chart . . . 12
3.2 Use Case UC1 - Sotto casi d'uso . . . 13
3.3 Use Case UC2 - Visualizzazione Point Series Chart . . . 15
3.4 Use Case UC2 - Sotto casi d'uso . . . 15
3.5 Use Case UC3 - Visualizzazione Importi Totali . . . 17
3.6 Use Case UC3 - Sotto casi d'uso . . . 17
3.7 Use Case UC4 - Scelta Contenuto Widget . . . 18
3.8 Use Case UC4 - Sotto casi d'uso . . . 19
3.9 Use Case UC5 - Visualizzazione Messaggio Errore Pagina . . . 21
3.10 Use Case UC5 - Sotto casi d'uso . . . 21
3.11 Use Case UC6 - Visualizzazione Pagina . . . 22
3.12 Use Case UC6 - Sotto casi d'uso . . . 23

4.1 Schema del design pattern MVC . . . 30
4.2 Schema del design pattern Dependency Injection . . . 31
4.3 Utilizzo di Inject in un componente .razor . . . 31
4.4 Utilizzo di Inject in una classe . . . 31
4.5 Suddivisone del progetto GalileoProgetti.Web.BaseBlazor . . . 32
4.6 Organizzazione della soluzione GalileoProgetti.Web.MatchBlazor . . . 33

5.1 Implementazione del metodo OnInitializedAsync . . . 36
5.2 Implementazione del componente ComponentLoader . . . 37
5.3 Utilizzo di ComponentLoader . . . 37
5.4 Visualizzazione widget della pagina Dati Generali . . . 37
5.5 Implementazione del metodo OnInitializedAsync . . . 38
5.6 Implementazione del metodo WidgetFactory . . . 38
5.7 Implementazione parziale del componente WidgetDonutChart . . . 39
5.8 Visualizzazione del loading di un widget . . . 39
5.9 Implementazione del metodo OnInitialized nella classe WidgetDonutChart 40
5.10 Implementazione del metodo Export . . . 40

5.11 Visualizzazione del widget contenente il donut chart 41
5.12 Visualizzazione errore di un widget 41
5.13 Implementazione parziale del componente WidgetPointSeriesChart . . 42
5.14 Visualizzazione del widget contenente il point series chart 43
5.15 Implementazione parziale di OneInfoWidget.razor 43
5.16 Visualizzazione del widget con l'elenco degli importi totali 44
5.17 Visualizzazione della scelta di ogni widget 44
5.18 Implementazione del metodo NewPopUp 45
5.19 Serializzazione dell'oggetto combo contenente gli id dei widget 45
5.20 Serializzazione della stringa dataJson 45
5.21 Visualizzazione di tre donut chart 46
5.22 Visualizzazione elenco importo totali, donut chart e point series chart 46
5.23 Esempio di utilizzo di GalDateEdit 46
5.24 Esempio di utilizzo di GalCheckBox 46
5.25 Esempio di utilizzo di GalTextEdit 47
5.26 Esempio di utilizzo di GalLookUp 47
5.27 Esempio di utilizzo di GalNumericEdit 47
5.28 Esempio di utilizzo di DxComboBox 47
5.29 Implementazione del componente GalGridNew 48
5.30 Implementazione parziale del componente NoteLibroSociale 49
5.31 Visualizzazione di una riga della griglia modificabile 49
5.32 Visualizzazione del form per inserire una nuova riga 49
5.33 Implementazione del metodo Grid_CustomGroup() 50
5.34 Implementazione del metodo Grid_CustomizeGroupValueDisplayText() 50
5.35 Implementazione del componente Strumenti Finanziari 50
5.36 Visualizzazione della griglia della pagina Strumenti Finanziari 51
5.37 Struttura di un file di risorse 51
5.38 Utilizzo di IStringLocalizer<T> 51
5.39 Esempio di accesso diretto al file di risorse 52

6.1 Visualizzazione dl test dei messaggi d'errore in un form 55
6.2 Esempio di testing 56

Elenco delle tabelle

3.1 Tabella del tracciamento dei requisti funzionali 26
3.2 Tabella del tracciamento dei requisiti qualitativi 27
3.3 Tabella del tracciamento dei requisiti di vincolo 27

7.1 Tabella del tracciamento dei requisti funzionali soddisfatti 59
7.2 Tabella del tracciamento dei requisiti qualitativi soddisfatti 60
7.3 Tabella del tracciamento dei requisiti di vincolo soddisfatti 60

Capitolo 1

Introduzione

1.1 Presentazione dell'azienda

L'azienda *Galileo Network S.p.A*[1] mi ha dato la possibilità di eseguire le ore di tirocinio previste per il Corso di Informatica. L'azienda si occupa degli intermediari finanziari non bancari e, in particolare, dei confidi, per i servizi di *outsourcing*$_G$ informatico e consulenza. L'azienda è caratterrizata da diversi fattori:

- Esperienza: percorso che si sta sviluppando da più di 20 anni;
- Sistemi informativi avanzati: sviluppi e innovazioni continui di prodotti e soluzioni;
- Una differenziata gamma di prodotti: ampia scelta di soluzioni tecnologiche.

Le sedi dell'azienda sono tre: Padova, Modena e Vicenza, e tutte sono certificate *ISO 9001:2015*: si tratta della norma internazionale per i Sistemi di Gestione per la Qualità *(SGQ*$_G$*)*, pubblicata dall'*International Organization for Standardization (ISO)*.

1.1.1 Confidi

Confidi[2] è l'acronimo di "*Consorzio di garanzia colletiva dei fidi*" ed è un consorzio italiano che si impegna a fornire alle aziende, che fanno parte del consorzio, le garanzie verso il sistema bancario necessarie per agevolarle nell'accesso ai finanziamenti a breve, medio e lungo termine, destinati alle attività economiche e produttive. Esistono due tipologie di confidi sottoposti a regime di controllo differenziati: i confidi minori (volume di attività finanziarie inferiori ai 150 milioni) e i confidi maggiori (volume di attività finanziarie superiore ai 150 milioni).

1.1.2 Match!

Si tratta di una delle soluzioni dell'azienda e tramite l'utilizzo di tecnologie innovative, permette la pressochè totale automazione dell'operatività ordinaria. La soluzione *Match!*, a livello di codice, è suddivisa nel modo seguente:

[1] *Galileo Network S.p.A.* URL: https://www.galileonetwork.eu/about/.

[2] *Consorzio fidi.* URL: https://www.confindustria.vr.it/confindustria/verona/istituzionale.nsf/($linkacross)/33A9F829CAB8A45AC12580F200484463/$file/1-17%20pillola%20finanza%20Confidi.pdf?openelement.

- Stampe: questa non è stata un'area di mio interesse;
- *WebApi*: è un'applicazione web senza UI_{G} che espone API_{G}. Viene usata da alcuni microservizi e dalla soluzione $Blazor_{\text{G}}$;
- *Login*: si tratta dell'applicazione responsabile dell'accesso;
- *Match o PraticoWeb*: è l'applicazione *Match!* attuale.

1.2 Presentazione dello stage

Lo stage ha avuto una durata di ..ore, suddvidendole 8 ore al giorno lavorative per 5 giorni a settimana. Ogni settimana sono stati svolti 3 giorni in sede e 2 giorni in *smart working*.

1.3 L'idea

L'origine dell'idea di questo progetto, nasce perchè l'azienda sente la necessità di creare un'applicazione moderna analoga a *Match!* in grado di sostituirla. Questo perchè *Match!* utilizza tecnologie datate e pecca di efficienza, soprattutto in termini di tempo: infatti solo per accederci, prima di visualizzare la pagina principale, l'attesa è di parecchi secondi (maggiore di 15s); ovviamente questo problema si presenta nuovamente quando si prova a navigare nelle altre pagine e, in particolar modo, quando vengono aggiunti, rimossi, aggiornati dati presenti nei form.
L'azienda, soprattutto il mio tutor, analizzando le mie aspettative e cercando un punto di incontro, aveva deciso di improntare lo stage nel campo dello sviluppo web, area che appunto mi appassiona e stimola a studiarla accuratamente.

1.3.1 Obiettivo

L'obiettivo di questo stage è stato quello di iniziare a progettare e sviluppare un'applicazione web in grado di sostuire *Match!* ma, data la vastità di informazioni contenute in essa in quantità di pagine, dettagli ed ecc, tale nuova applicazione non avrebbe mai potuto essere terminata a stage concluso: infatti a questo progetto non ho lavorato da sola, ma insieme al mio tutor.
Più precisamente mi è stato dato l'incarico di sviluppare delle pagine web che riassumessero i dati di una specifica azienda, i quali possono essere di tipo anagrafici o relativi ai tipi di finanziamenti con annessi i costi.

1.4 Organizzazione del testo

La tesi è organizzata nel seguente modo:

Il secondo capitolo descrive nello specifico tutte le tecnologie che sono state utilizzate per realizzare l'applicazione web;

Il terzo capitolo riporta i requisiti che l'applicazione deve soddisfare, utilizzando Use Case$_{\text{G}}$ e tabelle;

Il quarto capitolo approfondisce la progettazione dell'applicazione web, analizzando le scelte che sono state prese;

Il quinto capitolo descrive la fase di sviluppo dell'applicazione;

Il sesto capitolo riporta i test che sono stati fatti per verificare la correttezza e funzionalità di alcune parti dell'applicazione;

Il settimo capitolo riassume brevemente il lavoro svolto, traendo conclusioni sul prodotto finale.

Riguardo la stesura del testo, relativamente al documento sono state adottate le seguenti convenzioni tipografiche:

- Vengono definiti nel glossario, situato alla fine del presente documento, gli acronimi e i termini ambigui o di uso non comune;
- Per i termini che sono stati riportati nel glossario, viene utilizzato il carattere corsivo e aggiunta la lettera G a pedice della parola (Esempio: $parola_{\text{G}}$);
- I termini in lingua straniera o facenti parti del gergo tecnico sono evidenziati con il carattere corsivo (Esempio: *parola*).

Capitolo 2

Tecnologie

Per realizzare l'applicazione web, in tutti i suoi aspetti, quindi sia dal lato sviluppatore sia dal lato di comunicazione tra noi membri, mi è stato prestato un computer portatile aziendale con già installate le diverse tecnologie che si sono utilizzate, le quali verranno descritte in dettaglio nei paragrafi successivi.

2.1 Blazor

Blazor[1] è un *framework Web* gratuito e *open source* utilizzato per sviluppare applicazioni a pagina singola, tramite i linguaggi C# e *HTML* e *CSS*. Il nome deriva dalla fusione di due parole: *Browser* e *Razor* (componente che genera la visualizzazione *HTML*).
Il codice sorgente è di proprietà di *The .NET Foundation*, organizzazione che supporta progetti *open source* basati sul *framework .NET*. Blazor non richiede alcun tipo di *plug-in* installato a livello *client* per essere eseguito.
La figura 2.1 riporta il logo di *Blazor*.

Figura 2.1: Logo di Blazor

2.1.1 ASP.NET Core

ASP.NET Core è un *framework open source*, ad alte prestazioni per la creazione di app moderne connesse ad *Internet*. I vantaggi che offre sono i seguenti:

- Strumenti che semplificano lo sviluppo del web moderno;
- *Pipeline* di richiesta *HTTP* leggera, a prestazione elevate e modulare;

[1] *Blazor, Server e WebAssembly.* URL: https://learn.microsoft.com/en-us/aspnet/core/blazor/hosting-models?view=aspnetcore-6.0.

- È *open source*;
- Inserimento delle dipendenze incorporato;
- Con *Blazor* permette di usare C# nel *broswer* insieme a *Javascript*, e permette di condividere la logica dell'app (scritta con *.NET*), sia sul lato *client* sia sul lato *server*.

2.1.2 Blazor Server

Blazor Server è una delle cinque edizioni di *Blazor*. Con *Blazor Server* l'applicazione viene eseguita nel server dall'interno di un'applicazione *ASP.NET Core*. La gestione degli eventi e le chiamate *Javascript* vengono gestite tramite una connessione *SignalR*. Quest'ultima si occupa della gestione della connessione e consente di trasmettere messaggi a tutti i *client* connessi contemporaneamente.
Nel *client*, lo script *blazor.server.js* stabilisce la connessione *SignalR* con il *server*. Questo tipo di connessione tra *client e server* è persistente.

2.1.3 Blazor WebAssembly

Blazor WebAssembly è un'altra edizione di *Blazor* ed è quella che è stata utilizzata per la realizzazione dell'applicazione web. Le applicazioni a pagina singola vengono scaricate nel *browser* del *client* prima dell'esecuzione. Gli aggiornamenti della UI_G e la gestione degli eventi si verificano all'interno dello stesso processo. Un'applicazione *client* ospitata può interagire con l'applicazione *server back-end* sulla rete, usando un'ampia gamma di *framework* e protocolli di messaggistica, ad esempio API_G *Web e SignalR*.
Lo script *balzor.webassembly.js* viene fornito dal *framework* e gestisce il *download* del *runtime .NET*, dell'applicazione e delle sue dipendenze e dell'inizializzazione del *runtime* per eseguire l'applicazione.
La dimensione del *download* è maggiore rispetto a quello di *Blazor Server*, tuttavia questo tipo di applicazione gode di tempi rapidi di risposta, motivo per cui si è deciso di utilizzare tale *framework*. Un altro motivo che ha spinto a scegliere *Blazor* è la possibilità di usare il linguaggio *C#*, già conosciuto ai membri dell'azienda.
Più precisamente, i passi che vengono svolti durante la compilazione di un'app *Blazor WebAssembly* sono i seguenti (riassunti nella figura 2.2):

- I *file Razor* e contenenti il codice vengono compilati in *assembly .NET*;
- Il *runtime .NET* viene scaricato nel *browser*;
- Il *Blazor WebAssembly runtime* usa l'interoperabilità *Javascript* per gestire la manipolazione DOM_G e le chiamate API_G.

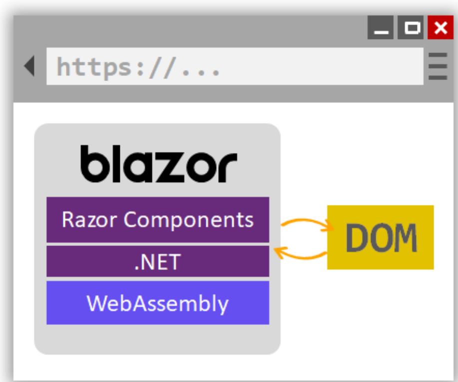


Figura 2.2: Compilazione di un'app Blazor WebAssembly

2.2 SQL Server Management Studio

SQL Server Management Studio, abbreviato in *SSMS*[2], è un ambiente integrato per la gestione di qualsiasi infrastruttura *SQL* sviluppata da *Microsoft*. È stato lanciato per la prima volta con *Microsoft SQL Server 2005*; la versione che è stata utilizzata per il progetto è la 18, rilasciata a Giugno 2022. *SSMS* fornisce strumenti per configurare, monitorare e amministrare istanze di *SQL Server* e *database*: infatti viene utilizzato per interrogare, progettare e gestire essi.
La figura 2.3 riporta il logo di *SQL Server Management Studio*.

Figura 2.3: Logo di SQL Server Manager Studio

2.3 Visual Studio 2022

Visual Studio è un'*IDE (Integrated Development Environment)* di *Microsoft* e viene utilizzato per sviluppare *GUI (Graphical User Interface), console, applicazioni web, app mobile*, ecc.. Non è un'*IDE* specifico per un linguaggio, in quanto è possibile utilizzarlo per scrivere codice in *C#, C++, Python, Javascript*, ecc.., in totale fornisce supporto per 36 linguaggi di programmazione differenti.
È strutturato nel seguente modo:

[2] *SSMS*. URL: https://learn.microsoft.com/en-us/sql/ssms/download-sql-server-management-studio-ssms?view=sql-server-ver16.

1. *Editor* di codice: l'utente scriverà in questa parte il codice;

2. Finestra di *ouput*: qui verranno mostrati le informazioni di debug e gli avvisi del compilatore, con relativi *warning* ed elenco errori;

3. Esplora soluzioni: mostra le soluzioni del proprio progetto, con annesse cartelle e file;

4. Proprietà: vengono mostrati dettagli e informazioni aggiuntive sulle parti selezionate.

La figura 2.4 riporta il logo di *Visual Studio 2022*.

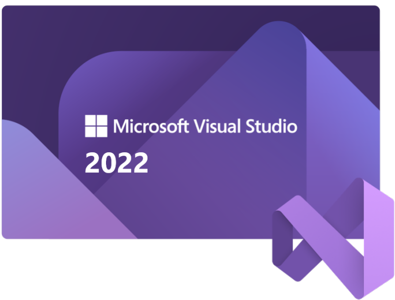


Figura 2.4: Logo di Visual Studio 2022

2.4 GitLab

GitLab è una piattaforma web dedicata alla gestione di *repository Git*. Si tratta di uno dei sistemi di controllo della versione per la gestione del codice nello sviluppo *software*. Infatti, per i lavori in *team* offre il vantaggio di poter scrivere il codice in contemporanea ma in modo autonomo, effettuando modifiche, le quali possono essere tracciate e anche annullate se necessario. Sono consentite operazioni di:

- *Pull*: scaricare in locale il codice contenuto nel *repository*;

- *Push*, l'opposto: caricare sul *repository* le modifiche che sono state apportate in locale;

- *Merge*: unire le modifiche proposte rispetto al codice originale.

La figura 2.5 riporta il logo di *GitLab*.

Figura 2.5: Logo di GitLab

2.5 Microsoft Teams

Microsoft Teams è una piattaforma di comunicazione che permette di riunire gli utenti registrati in videoconferenze e in *chat*, individuali o di grupppo, condividendo un calendario per fissare riunioni e appuntamenti. Durante il periodo di *smart working* tale programma è stato fondamentale per la collaborazione con il *tutor*.
La figura 2.6 riporta il logo di *Microsoft Teams*.

Figura 2.6: Logo di Microsoft Teams

2.6 Pulse Secure

Pulse consente connessioni autenticate sicure a risorse di rete protette su reti *LAN* e *WAN*. *Pulse Secure Client* semplifica la connettività della rete consentendo all'amministratore di rete di configurarne l'ambiente di accesso. Utilizzando una connessione sicura, il traffico di rete viene crittografato in modo che solo l'*endpoint* specifico e il *server* con cui sta comunicando possano comprenderne il contenuto.
Le impostazioni di sicurezza utilizzate da *Pulse Secure Client* quando stabilisce una connessione di rete vengono definite dall'amministratore di rete. Una connessione autenticata significa che l'*endpoint* e il *server* hanno entrambi accertato reciprocamente le loro identità. Per autenticarsi, il *server* utilizza un certificato di un'autorità di certificazione attendibile e l'*endpoint* presenta le credenziali di accesso al *server*. L'amministratore di rete può configurare l'ambiente in modo che l'autenticazione avvenga automaticamente senza richiedere alcuna informazione, anche se in genere viene chiesta l'immissione del nome utente e della *password* al momento della connessione. *Pulse Secure Client* visualizza inoltre nell'area di notifica un'icona che segnala lo stato di connessione, consente di connettersi e disconnettersi da reti e riunioni e di accedere rapidamente all'interfaccia del programma.
Per poter lavorare in *smart working* è stato necessario l'utilizzo di questa tecnologia per connettersi alla VPN_G aziendale.
La figura 2.7 riporta il logo di *Pulse Secure*.

Figura 2.7: Logo di Pulse Secure

Capitolo 3

Analisi dei requisiti

3.1 Casi d'uso

Per lo studio dei casi di utilizzo del prodotto sono stati creati dei diagrammi. I diagrammi dei *casi d'uso* sono diagrammi di tipo UML_G dedicati alla descrizione delle funzioni o servizi offerti da un sistema, così come sono percepiti e utilizzati dagli attori che interagiscono col sistema stesso.

3.1.1 Attori

Le funzionalità create dell'applicazione riguardano solo utenti autenticati, quindi utenti che hanno effettuato il *login*, e per convenzione quando verrà scritto utente si intende *utente autenticato*. In questi casi, l'utente è un'azienda che decide di visionare le informazioni personali riguardo ad una relativa pagina. La pagina principale dell'applicazione è "Dati Generale".

3.1.2 Classificazione

Ogni *caso d'uso* verrà identificato nel seguente modo:

- **UC**: abbreviazione di $Use\ Case_G$;
- **[Codice]**: ad ogni UC_G viene assegnato un numero identificativo, diverso per tutti;
- **[CodiceFiglio]**: alcuni UC_G sono "figli" di un UC_G "padre", quindi hanno un secondo codice.

In conclusione la classificazione per ogni UC_G è la seguente: **UC [Codice].[CodiceFiglio]**

3.1.3 Descrizione

Per ogni *caso d'uso* vengono riportati i seguenti dettagli:

- **Attori principali:** sono gli attori che interagiscono con lo specifico UC_G;
- **Descrizione**: è la descrizione dello UC_G in questione;
- **Scenario:** è la successione degli eventi;

- **Precondizioni:** si tratta delle condizioni che devono essere vere prima dell'esecuzione dello UC_G;
- **Postcondizioni:** si tratta delle condizioni che devono essere vere subito dopo l'esecuzione dello UC_G;
- **Estensioni:** eventuali estensioni di uno UC_G(opzionale, non sempre presente).

3.2 Elenco Casi d'Uso

UC1: Visualizzazione Donut Chart

La figura 3.1 riporta il caso d'uso relativo alla visualizzazione del *donut chart*.

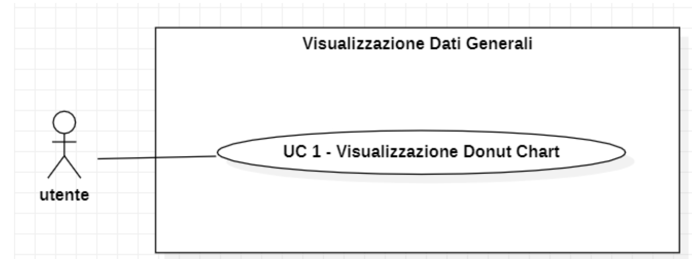


Figura 3.1: Use Case UC1 - Visualizzazione Donut Chart

Attori Principali: Utente autenticato.

Descrizione: L'utente quando si trova nella pagina "Dati Generali" deve essere in grado di visualizzare il *widget* contenente un *donut chart* con informazioni riguardo ad uno degli importi. Gli importi possono essere: totale importo richiesto, totale importo deliberato, totale importo erogato, totale impegno, totale residuo, totale esposizione, totale rischio, totale sofferenza di firma, totale sofferenza di cassa, totale impegno reale e totale disponibilità.

Scenario:

1. L'utente naviga nell'applicazione;
2. L'utente clicca sulla pagina Dati Generali;
3. L'utente visualizza il *Donut Chart*.

Precondizioni: L'utente vuole visualizzare il *donut chart* con un importo.

Postcondizioni: L'utente visualizza il *donut chart* con un importo.

Estensioni:

1. L'utente deve essere in grado di visualizzare un *donut chart* di *default* se non ha mai scelto quale importo visualizzare; UC1.1
2. L'utente deve essere in grado di esportare il *donut chart*; UC1.2

3. L'utente deve essere in grado di visualizzare il *loading* del *widget* contenente il *donut chart* se esso è in fase di caricamento; UC1.3

4. L'utente deve essere in grado di visualizzare un messaggio d'errore se ci sono stati problemi durante il carimento del *widget*. UC1.4

La figura 3.2 riporta i sotto casi d'uso relativi alla visualizzazione del *donut chart*.

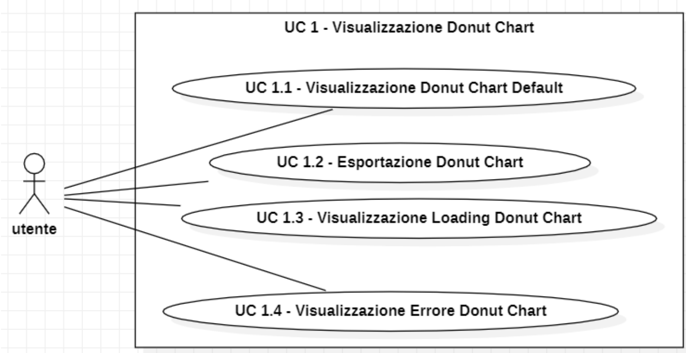


Figura 3.2: Use Case UC1 - Sotto casi d'uso

UC1.1: Visualizzazione Donut Chart Default

Attori Principali: Utente autenticato.

Descrizione: L'utente è la prima volta che entra nell'applicazione e non ha mai scelto quale importo visualizzare, quindi il *widget* non può essere vuoto e deve visualizzare un importo di *default*.

Scenario:

1. L'utente ha effetuato l'acceso per la prima volta;
2. L'utente naviga nell'applicazione;
3. L'utente clicca sulla pagina Dati Generali;
4. L'utente visualizza il *donut chart* con un importo di *defualt*.

Precondizioni: L'utente vuole visualizzare il *donut chart*.

Postcondizioni: L'utente visualizza il *donut chart* con l'importo di default.

UC1.2: Esportazione Donut Chart

Attori Principali: Utente autenticato.

Descrizione: L'utente dopo aver visualizzato il *donut chart* vuole scaricare la foto di esso.

Scenario:

1. L'utente clicca sulla pagina Dati Generali;

2. L'utente visualizza il *donut chart*;

3. L'utente clicca l'icona che permette di esportarlo.

Precondizioni: L'utente visualizza il *donut chart* e vuole esportarlo.
Postcondizioni: L'utente esporta il *donut chart* con i dati e la legenda.

UC1.3: Visualizzazione Loading Donut Chart

Attori Principali: Utente autenticato.
Descrizione: L'utente vuole visualizzare il *widget* contenente il *donut chart*, ma esso è in fase di caricamento.
Scenario:

1. L'utente clicca sulla pagina Dati Generali per visualizzare il *donut chart*;

2. L'utente visualizza il *loading* del *widget* che contiene il *donut chart*.

Precondizioni: L'utente vuole visualizzare il *widget* contenente il *donut chart*.
Postcondizioni: L'utente visualizza il *loading* del *widget* che contiene il *donut chart*.

UC1.4: Visualizzazione Errore Donut Chart

Attori Principali: Utente autenticato.
Descrizione: L'utente vuole visualizzare il *widget* contenente il *donut chart*, ma durante il suo carimento si verifica un errore.
Scenario:

1. L'utente clicca sulla pagina Dati Generali per visualizzare il *donut chart*;

2. L'utente visualizza un messaggio d'errore nel *widget* che contiene il *donut chart*.

Precondizioni: L'utente vuole visualizzare il *widget* contenente il *donut chart*.
Postcondizioni: L'utente visualizza un messaggio d'errore nel *widget* che contiene il *donut chart*.

UC2: Visualizzazione Point Series Chart

La figura 3.3 riporta il caso d'uso relativo alla visualizzazione del *point series chart*.

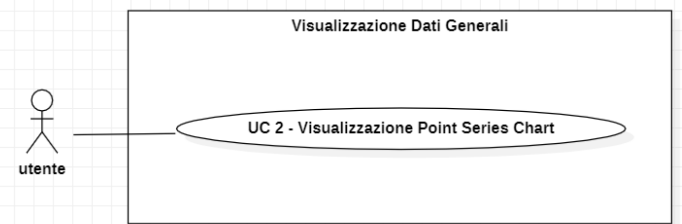


Figura 3.3: Use Case UC2 - Visualizzazione Point Series Chart

Attori Principali: Utente autenticato.

Descrizione: L'utente quando si trova nella pagina "Dati Generali" deve essere in grado di visualizzare il *widget* contenente il *point series chart*.

Scenario:

1. L'utente naviga nell'applicazione;
2. L'utente clicca sulla pagina Dati Generali;
3. L'utente visualizza il *point series chart*.

Precondizioni: L'utente vuole visualizzare il *point series chart*.

Postcondizioni: L'utente visualizza il *point series chart*.

Estensioni:

1. L'utente deve essere in grado di esportare il *point series chart*; UC2.1
2. L'utente deve essere in grado di visualizzare il *loading* del *widget* contenente il *point series chart* se esso è in fase di caricamento; UC2.2
3. L'utente deve essere in grado di visualizzare un messaggio d'errore se ci sono stati problemi durante il caricamento del *widget*. UC2.3

La figura 3.4 riporta i sotto casi d'uso relativi alla visualizzazione del *point series chart*.

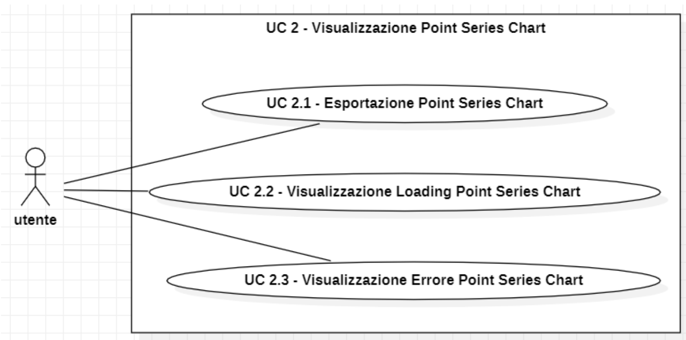


Figura 3.4: Use Case UC2 - Sotto casi d'uso

UC2.1: Esportazione Point Series Chart

Attori Principali: Utente autenticato.

Descrizione: L'utente dopo aver visualizzato il *point series chart* vuole scaricare la foto di esso.

Scenario:

1. L'utente clicca sulla pagina Dati Generali;
2. L'utente visualizza il *point series chart*;
3. L'utente clicca l'icona che permette di esportare il *point series chart.*

Precondizioni: L'utente visualizza il *point series chart* e vuole esportarlo.

Postcondizioni: L'utente esporta il *point series chart* con i dati e la legenda.

UC2.2: Visualizzazione Loading Point Series Chart

Attori Principali: Utente autenticato.

Descrizione: L'utente vuole visualizzare il *widget* contenente il *point series chart*, ma esso è in fase di caricamento.

Scenario:

1. L'utente clicca sulla pagina Dati Generali per visualizzare il *point series chart*;
2. L'utente visualizza il *loading* del *widget* che contiene il *point series chart.*

Precondizioni: L'utente vuole visualizzare il *widget* contenente il *point series chart.*

Postcondizioni: L'utente visualizza il *loading* del *widget* che contiene il *point series chart.*

UC2.3: Visualizzazione Errore Point Series Chart

Attori Principali: Utente autenticato.

Descrizione: L'utente vuole visualizzare il *widget* contenente il *point series chart*, ma durante il suo caricamento si verifica un errore.

Scenario:

1. L'utente clicca sulla pagina Dati Generali per visualizzare il *point series chart*;
2. L'utente visualizza un messaggio d'errore nel *widget* che contiene il *point series chart.*

Precondizioni: L'utente vuole visualizzare il *widget* contenente il *point series chart.*

Postcondizioni: L'utente visualizza un messaggio d'errore nel *widget* che contiene il *point series chart.*

UC3: Visualizzazione Importi Totali

La figura 3.5 riporta il caso d'uso relativo alla visualizzazione degli importi totali.

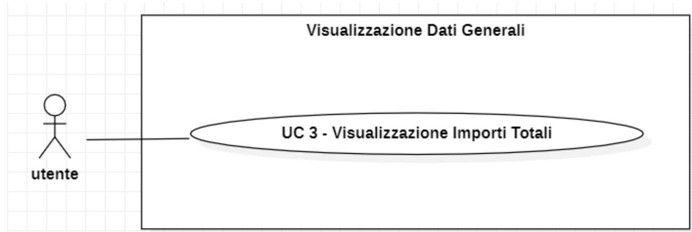


Figura 3.5: Use Case UC3 - Visualizzazione Importi Totali

Attori Principali: Utente autenticato.

Descrizione: L'utente quando si trova nella pagina "Dati Generali" deve essere in grado di visualizzare il *widget* contenente un elenco di tutti gli importi.

Scenario:

1. L'utente naviga nell'applicazione;
2. L'utente clicca sulla pagina Dati Generali;
3. L'utente visualizza gli importi totali.

Precondizioni: L'utente vuole visualizzare il *widget* contenente l'elenco degli importi totali.

Postcondizioni: L'utente visualizza il *widget* con l'elenco degli importi totali.

Estensioni:

1. L'utente deve essere in grado di visualizzare il *loading* del *widget* contenente l'elenco degli importi totali se esso è in fase di caricamento; UC3.1
2. L'utente deve essere in grado di visualizzare un messaggio d'errore se ci sono stati problemi durante il caricamento del *widget*. UC3.2

La figura 3.6 riporta i sotto casi d'uso relativi alla visualizzazione degli importi totali.

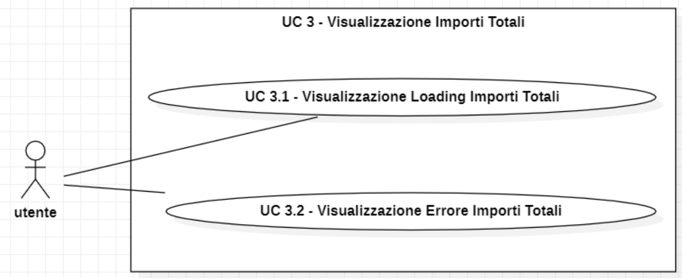


Figura 3.6: Use Case UC3 - Sotto casi d'uso

UC3.1: Visualizzazione Loading Importi Totali

Attori Principali: Utente autenticato.

Descrizione: L'utente vuole visualizzare il *widget* contenente l'elenco degli importi totali, ma esso è in fase di caricamento.

Scenario:

1. L'utente clicca sulla pagina Dati Generali per visualizzare gli importi totali;
2. L'utente visualizza il *loading* del *widget* che contiene gli importi totali.

Precondizioni: L'utente vuole visualizzare il *widget* contenente l'elenco degli importi totali.

Postcondizioni: L'utente visualizza il *loading* del *widget* che contiene l'elenco degli importi totali.

UC3.2: Visualizzazione Errore Importi Totali

Attori Principali: Utente autenticato.

Descrizione: L'utente vuole visualizzare il *widget* contenente l'elenco degli importi totali, ma durante il suo caricamento si verifica un errore.

Scenario:

1. L'utente clicca sulla pagina Dati Generali per visualizzare l'elenco degli importi totali;
2. L'utente visualizza un messaggio d'errore nel *widget* che contiene l'elenco gli importi totali.

Precondizioni: L'utente vuole visualizzare il *widget* contenente l'elenco degli importi totali.

Postcondizioni: L'utente visualizza un messaggio d'errore nel *widget* che dovrebbe contenere l'elenco degli importi totali.

UC4: Scelta Contenuto Widget

La figura 3.7 riporta il caso d'uso relativo alla scelta del contenuto di ogni *widget*.

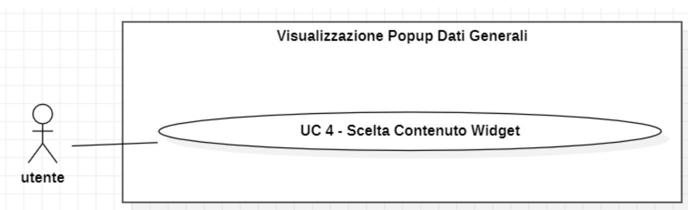


Figura 3.7: Use Case UC4 - Scelta Contenuto Widget

Attori Principali: Utente autenticato.

Descrizione: L'utente vuole scegliere quale tipo di informazione visualizzare per ogni *widget*.

Scenario:

1. L'utente clicca sulla pagina Dati Generali;
2. L'utente visualizza i tre *widget*;
3. L'utente clicca l'icona con il "+";
4. L'utente, dopo l'apertura di un *popup*, sceglie cosa visualizzare per ogni *widget*.

Precondizioni: L'utente vuole decidere cosa visualizzare per ogni *widget*.

Postcondizioni: L'utente ha scelto il contenuto di ogni *widget*.

Estensioni:

1. L'utente deve essere in grado di scegliere cosa visualizzare nel *widget* 1; UC4.1
2. L'utente deve essere in grado di scegliere cosa visualizzare nel *widget* 2; UC4.2
3. L'utente deve essere in grado di scegliere cosa visualizzare nel *widget* 3; UC4.3

La figura 3.8 riporta i sotto casi d'uso relativi alla scelta del contenuto di ogni *widget*.

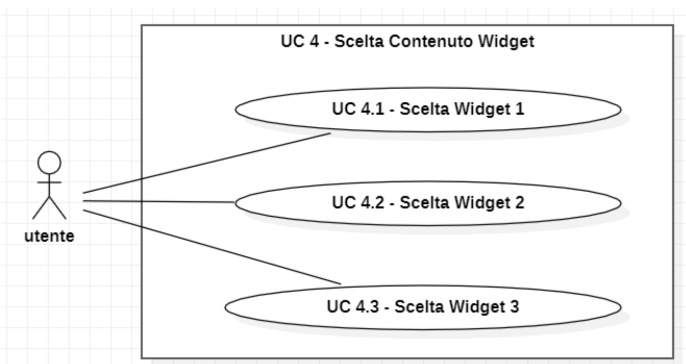


Figura 3.8: Use Case UC4 - Sotto casi d'uso

UC4.1: Scelta Widget 1

Attori Principali: Utente autenticato.

Descrizione: L'utente vuole decidere cosa visualizzare nel *widget* 1.

Scenario:

1. L'utente clicca sulla pagina Dati Generali;
2. L'utente visualizza i tre *widget*;

3. L'utente clicca l'icona con il "+";
4. L'utente, dopo l'apertura di un *popup*, sceglie cosa visualizzare per il *widget* 1 tra le possibili proposte.

Precondizioni: L'utente vuole scegliere cosa visualizzare nel *widget* in posizione 1.

Postcondizioni: L'utente ha scelto cosa visualizzare nel *widget* in posizione 1.

UC4.2: Scelta Widget 2

Attori Principali: Utente autenticato.

Descrizione: L'utente vuole decidere cosa visualizzare nel *widget* 2.

Scenario:

1. L'utente clicca sulla pagina Dati Generali;
2. L'utente visualizza i tre *widget*;
3. L'utente clicca l'icona con il "+";
4. L'utente, dopo l'apertura di un *popup*, sceglie cosa visualizzare per il *widget* 2 tra le possibili proposte.

Precondizioni: L'utente vuole scegliere cosa visualizzare nel *widget* in posizione 2.

Postcondizioni: L'utente ha scelto cosa visualizzare nel *widget* in posizione 2.

UC4.3: Scelta Widget 3

Attori Principali: Utente autenticato.

Descrizione: L'utente vuole decidere cosa visualizzare nel *widget* 3.

Scenario:

1. L'utente clicca sulla pagina Dati Generali;
2. L'utente visualizza i tre *widget*;
3. L'utente clicca l'icona con il "+";
4. L'utente, dopo l'apertura di un *popup*, sceglie cosa visualizzare per il *widget* 3 tra le possibili proposte.

Precondizioni: L'utente vuole scegliere cosa visualizzare nel *widget* in posizione 3.

Postcondizioni: L'utente ha scelto cosa visualizzare nel *widget* in posizione 3.

UC5: Visualizzazione Messaggio Errore Pagina

La figura 3.9 riporta il caso d'uso relativo alla visualizzazione di un messaggio d'errore della pagina cercata.

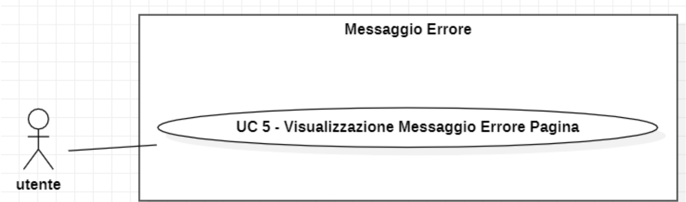


Figura 3.9: Use Case UC5 - Visualizzazione Messaggio Errore Pagina

Attori Principali: Utente autenticato.
Descrizione: L'utente vuole navigare in una pagina ma compare un messaggio d'errore.

Scenario:

1. L'utente naviga nell'applicazione;
2. L'utente cerca una determinata pagina;
3. L'utente visualizza un messaggio d'errore.

Precondizioni: L'utente vuole navigare in una certa pagina.
Postcondizioni: L'utente visualizza un messaggio d'errore.
Estensioni:

1. L'utente deve essere in grado di visualizzare un messaggio d'errore se la pagina cercata non è stata trovata; UC5.1
2. L'utente deve essere in grado di visualizzare un messaggio d'errore se non ha i permessi per accedere alla pagina cercata. UC5.2

La figura 3.10 riporta i sotto casi d'uso relativi alla visualizzazione di un messaggio d'errore della pagina cercata.

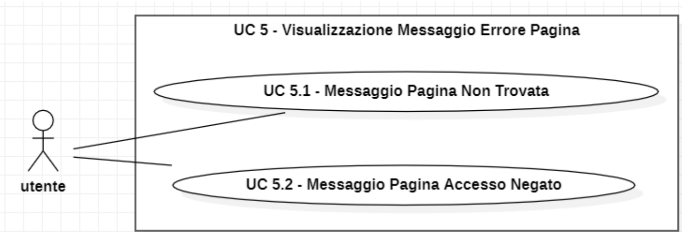


Figura 3.10: Use Case UC5 - Sotto casi d'uso

UC5.1: Messaggio Pagina Non Trovata

Attori Principali: Utente autenticato.

Descrizione: L'utente prova a navigare nell'applicazione ma la pagina cercata non è raggiungibile perchè l'*url* inserito è scorretto o perchè è stata rimossa.

Scenario:

1. L'utente naviga nell'applicazione;
2. L'utente non trova la pagina cercata.

Precondizioni: L'utente vuole navigare in una certa pagina.

Postcondizioni: L'utente visualizza la pagina d'errore di *defualt* con un messaggio.

UC5.2: Messaggio Pagina Accesso Negato

Attori Principali: Utente autenticato.

Descrizione: L'utente prova a navigare nell'applicazione ma la pagina cercata non è raggiungibile in quanto non ha i permessi necessari per accederci.

Scenario:

1. L'utente naviga nell'applicazione;
2. L'utente non ha i permessi per navigare nella pagina cercata.

Precondizioni: L'utente vuole navigare in una certa pagina.

Postcondizioni: L'utente visualizza la pagina di accesso negato di *defualt*.

UC6: Visualizzazione Pagina

La figura 3.11 riporta il caso d'uso relativo alla visualizzazione di una pagina che non sia la principale.

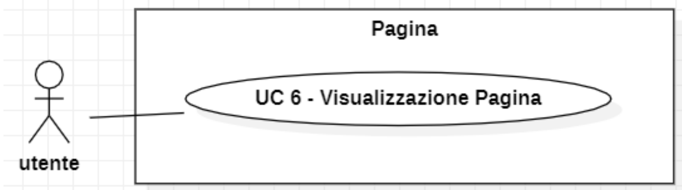


Figura 3.11: Use Case UC6 - Visualizzazione Pagina

Attori Principali: Utente autenticato.

Descrizione: L'utente vuole visitare una delle pagine presenti nell'applicazione.

Scenario:

1. L'utente naviga nell'applicazione;
2. L'utente clicca nella barra laterale di navigazione per scegliere quale pagina visitare;

3. L'utente clicca sulla pagina che vuole visitare.

Precondizioni: L'utente vuole visitare una pagina dell'applicazione.

Postcondizioni: L'utente visualizza la pagina che ha scelto.

La figura 3.12 riporta i sotto casi d'uso relativi alla visualizzazione di una pagina che non sia la principale.

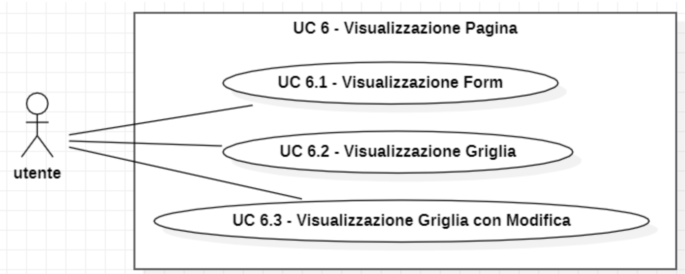


Figura 3.12: Use Case UC6 - Sotto casi d'uso

UC6.1: Visualizzazione Form

Attori Principali: Utente autenticato.

Descrizione: L'utente visita una pagina contenente un *form* relativo ad esso e con argomento la pagina scelta. In questo *form* l'utente può leggere solo alcuni valori, mentre altri può modificarli.

Scenario:

1. L'utente naviga nell'applicazione;
2. L'utente clicca nella barra laterale di navigazione per scegliere quale pagina visitare;
3. L'utente clicca sulla pagina che vuole visitare.

Precondizioni: L'utente vorrebbe visitare una pagina dell'applicazione.

Postcondizioni: L'utente visualizza la pagina scelta, la quale contiene un *form*.

UC6.2: Visualizzazione Griglia

Attori Principali: Utente autenticato.

Descrizione: L'utente visita una pagina contenente una griglia relativa ad esso e con valori che può solo leggere.

Scenario:

1. L'utente naviga nell'applicazione;
2. L'utente clicca nella barra laterale per scegliere quale pagina visitare;

3. L'utente clicca sulla pagina che vuole visitare.

Precondizioni: L'utente vuole visitare una pagina dell'applicazione.

Postcondizioni: L'utente visualizza la pagina scelta, la quale contiene una griglia in solo lettura.

UC6.3: Visualizzazione Griglia con Modifica

Attori Principali: Utente autenticato.

Descrizione: L'utente visita una pagina contenente una griglia relativa ad esso, con valori che possono essere parzialmente modificati. L'utente infatti è in grado di inserire, modificare e rimuovere una riga.

Scenario:

1. L'utente naviga nell'applicazione;
2. L'utente clicca nella barra laterale per scegliere quale pagina visitare;
3. L'utente clicca sulla pagina che vuole visitare.

Precondizioni: L'utente vuole visitare una pagina dell'applicazione.

Postcondizioni: L'utente visualizza la pagina scelta, la quale contiene una griglia parzialmente modificabile.

3.3 Tracciamento dei requisiti

3.3.1 Classificazione

Da un'attenta analisi dei requisiti e degli *use case*$_G$ effettuata sul progetto è stata stilata la tabella che traccia i requisiti in rapporto ai *casi d'uso*.
Sono stati individuati diversi tipi di requisiti e si è quindi fatto utilizzo di un codice identificativo per distinguerli.
Ogni requisito è classificato nel seguente modo:

R[Tipologia][Importanza][Codice]

dove:

- **Tipologia:** può essere:
  - **F:** si riferisce ad un requisito funzionale;
  - **Q:** si riferisce ad un requisito qualitativo;
  - **V:** si riferisce ad un requisito di vincolo.
- **Importanza:** può essere:
  - **O:** si riferisce ad un requisito obbligatorio;
  - **D:** si riferisce ad un requisito desiderabile;
  - **F:** si riferisce ad un requisito facoltativo.
- **Codice:** si riferisce ad un codice identificativo univoco associato ad ogni requisito.

3.3.2 Elenco Requisiti

Tabella del tracciamento dei requisiti funzionali

Nella tabella 3.1 sono elencati i requisiti funzionali.

Requisito	Descrizione	Use Case
RFO1	L'utente per poter interagire con l'applicazione deve essere autenticato	-
RFO2	L'utente deve poter visualizzare il *widget* contenente il *donut chart* con uno degli importi	UC 1
RFO3	L'utente deve essere in grado di esportare il *donut chart* in formato *.png*	UC 1.2
RFO4	L'utente deve visualizzare un messaggio d'errore se durante il caricamento del *donut chart* c'è stato un problema	UC 1.4
RFO5	L'utente deve poter visualizzare il *widget* contenente il *point series chart*	UC 2
RFO6	L'utente deve essere in grado di esportare il *point series chart* in formato *.png*	UC 2.1
RFO7	L'utente deve visualizzare un messaggio d'errore se durante il caricamento del *point series chart* c'è stato un problema	UC 2.3
RFO8	L'utente deve poter visualizzare il *widget* contenente gli importi totali	UC 3
RFO9	L'utente deve visualizzare un messaggio d'errore se durante il caricamento degli importi totali c'è stato un problema	UC 3.2
RFO10	L'utente deve poter scegliere il contenuto di ognuno dei tre *widget* tra le scelte presenti	UC 4
RFO11	L'utente deve visualizzare un messaggio se la pagina che ha cercato non esiste o è stata rimossa	UC 5.1
RFO12	L'utente deve visualizzare un messaggio se non ha i permessi per navigare nella pagina cercata	UC 5.2
RFO13	L'utente deve poter visualizzare e modificare alcuni valori dei *form*	UC 6.1
RFO14	L'utente deve poter visualizzare la griglia ed inserire, modificare ed eliminare una riga da essa	UC 6.2, UC 6.3
RFD1	L'utente deve visualizzare il *loading* del *donut chart* se esso è in fase di caricamento	UC 1.3
RFD2	L'utente deve visualizzare il *loading* del *point series chart* se esso è in fase di caricamento	UC 2.2
RFD3	L'utente deve visualizzare il *loading* degli importi totali se essi sono in fase di caricamento	UC 3.1
RFF1	L'utente deve visualizzare il *loading* all'interno della griglia, se essa è in fase di caricamento	-

Tabella 3.1: Tabella del tracciamento dei requisti funzionali

Tabella del tracciamento dei requisiti qualitativi

Nella tabella 3.2 sono elencati i requisiti qualitativi.

Requisito	Descrizione	Use Case
RQO1	Le pagine implementate devono essere traducibili sia in italiano sia in inglese	-
RQD1	Deve essere effettuato un test per i *form* implementati	-

Tabella 3.2: Tabella del tracciamento dei requisiti qualitativi

Tabella del tracciamento dei requisiti di vincolo

Nella tabella 3.3 sono elencati i requisiti di vincolo.

Requisito	Descrizione	Use Case
RVO1	Per la realizzazione dell'applicazione deve essere utilizzato il *framework Blazor v. 3.2.0*	-
RVO2	Tramite l'utilizzo di *Blazor* deve essere utilizzato *ASP .NET 6*	-
RVO3	L'applicazione deve essere supportato nelle versioni recenti di *Google Chrome, Microsoft Edge e Mozilla Firefox*	-

Tabella 3.3: Tabella del tracciamento dei requisiti di vincolo

Capitolo 4

Progettazione

4.1 Design Pattern utilizzati

In questa sezione vengono illustrati e descritti i *design patterns* che si sono utilizzati per la realizzazione dell'applicazione.

4.1.1 Model-View-Controller

Model-View-Controller, abbreviato in *MVC*, è un *design patterns architetturale* che divide *business logic* e interfaccia. Consiste in un'architettura composta da tre parti differenti:

- *Model*: è il modello, ovvero la struttura dati dinamica dell'applicazione, indipendente dall'interfaccia utente. Si occupa della gestione diretta dei dati, della logica e delle regole dell'applicazione;

- *View*: è la vista e si occupa della visualizzazione dei dati, ad esempio un grafico, una tabella, un diagramma, ecc;

- *Controller*: gestisce gli *input* con opportune chiamate al modello o alla vista. Le richieste di un sito web vengono indirizzate al *controller*, il quale è responsabile di lavorare con il modello per eseguire azioni e/o recuperare dati. Il *controller*, infatti, sceglie la vista da visualizzare e le fornisce il modello. La vista esegue il *rendering*$_G$ della pagina finale da visualizzare, in base ai dati nel modello.

Nella figura 4.1 viene riassunto lo schema del *design pattern MVC*, come quanto appena riportato.

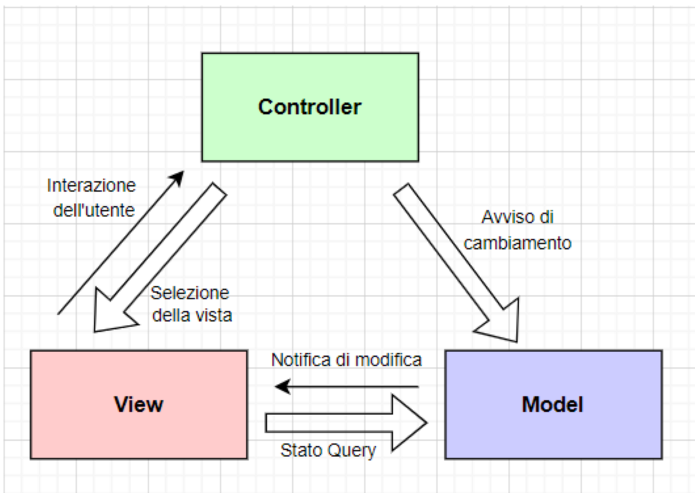


Figura 4.1: Schema del design pattern MVC

Motivo della scelta del pattern MVC

Si è deciso di utilizzare questo *design pattern* perchè mantenere queste tre parti logicamente separate offre dei vantaggi nella gestione del codice, infatti favorisce lo sviluppo, il *test* e la manutenzione di ciascuna parte indipendentemente dall'altra.
Il funzionamento di un'applicazione web è il seguente[1]: il *client* inoltra la richiesta ad un *server* per una pagina *HTML*; il *server* ospita un'applicazione scritta in un linguaggio di programmazione che preleva i dati da un *database*, elaborandoli e restituendoli poi al *client* in formato *HTML*. Se provassimo a scrivere un'applicazione consistente in una singola pagina web che svolga tutti i compiti, è evidente che unire tutte queste operazioni in un unico blocco di codice creerebbe presto confusione e problemi, i quali comporterebbero a lunghe sessioni di *debug* per risolverli.

4.1.2 Dependency Injection

Dependency Injection, abbreviati in *DI*, è un *design pattern* che permette ad una classe di non dipendere direttamente da un'eventuale implementazione. Sfruttando le interfacce e il polimorfismo si può far interagire le due classi in maniera che il collegamento sia fatto tramite un'interfaccia, facendo quindi sì che il *design* sia semplice e facilmente manutenibile per l'evoluzione del sistema.
Blazor, tramite *ASP .NET*, funziona nel modo seguente:

- I servizi registrati nel *framework* possono essere inseriti direttamente nei componenti delle *Blazor app*;

- Le *app Blazor* definiscono e registrano servizi personalizzati e li rendono disponibili in tutta l'app tramite *DI*.

Nella figura 4.2 viene illustrato lo schema del *design pattern Dependency Injection*.

[1] *Design Pattern MVC*. URL: `https://www.html.it/pag/18299/il-pattern-mvc/`.

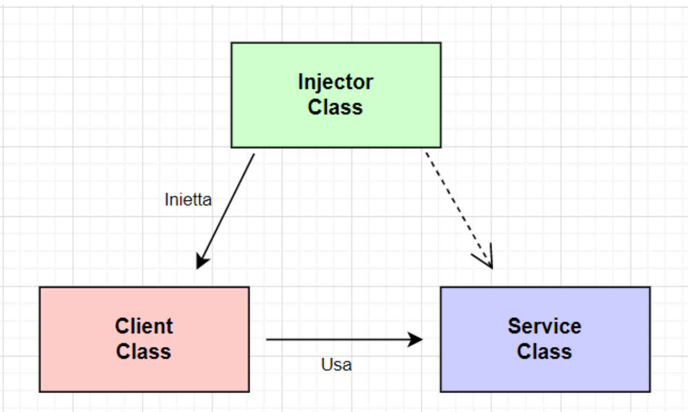


Figura 4.2: Schema del design pattern Dependency Injection

Dependency Injection in Blazor

In *Blazor* esistono due approcci diversi per indicare le dipendenze:

- Nel file *.razor* (esempio in figura 4.3) utilizzando la sintassi "***@inject*** ", nome dell'interfaccia e l'attributo;

```
@inject IJSRuntime js
```

Figura 4.3: Utilizzo di Inject in un componente .razor

- Nel file *.cs* (esempio in figura 4.4), ovvero la parte di codice *C#*, utilizzando la sintassi ***[Inject]***, nome dell'interfaccia e l'attributo.

```
[Inject]
IJSRuntime js { get; set; }
```

Figura 4.4: Utilizzo di Inject in una classe

Le dipendenze[2] vengono inserite dopo la creazione dell'istanza del componente *Blazor* e prima dell'esecuzione degli eventi del *ciclo di vita*$_G$ *OnInitialized o OnInitializedAsync*. Ciò significa che non possiamo sovrascrivere il costruttore del nostro componente e utilizzare quelle dipendenze da lì, ma possiamo usarle nei metodi *OnInitialized*.

4.1.3 AddScope in Blazor

Una dipendenza "*Scope*[3]" è simile a una dipendenza *Singleton*$_G$ in quanto *Blazor* inserirà la stessa istanza in ogni oggetto che dipende da essa; tuttavia, la differenza è che un'istanza "*Scope*" non è condivisa da tutti gli utenti.

[2] *Inject con Blazor*. URL: https://blazor-university.com/dependency-injection/injecting-dependencies-into-blazor-components/.

[3] *AddScope, Blazor*. URL: https://blazor-university.com/dependency-injection/dependency-lifetimes-and-scopes/scoped-dependencies/?.

Il primo oggetto che dipende da una dipendenza registrata "*Scope*" riceverà una nuova istanza di tale dipendenza e tale nuova istanza verrà memorizzata nella *cache* nel contenitore di iniezione. Da lì in poi, qualsiasi oggetto che richiede lo stesso tipo di dipendenza riceverà la stessa istanza memorizzata nella *cache*.
Quindi, al termine della richiesta, il contenitore non è più necessario e può essere sottoposto a *Garbage Collection*$_G$ insieme a tutte le istanze registrate "*Scope*" e transitorie che ha creato. Le istanze "*Scope*" ci consentono di registrare le dipendenze come istanza singola per utente anziché come istanza singola per applicazione.

4.2 Struttura della soluzione Blazor

La soluzione utilizzata per la realizzazione e implementazione dell'applicazione è *GalileoProgetti.Web.MatchBlazor.sln* ed è suddivisa nei seguenti sei progetti:

- *GalileoProgetti.Framework.NetGw*: contiene una cartella *DTO* che include le informazioni per i messaggi d'errore e per le operazioni di *input e output*;

- *GalileoProgetti.Web.BaseBlazor*: come raffigurato nella figura 4.5 il progetto corrente contiene le seguenti cartelle:

    · *js*: include i *file JavaScript*;

    · *BaseComponents*: include tutti i componenti base dell'applicazione. Infatti, per evitare di ripetere le stesse informazioni (quali valori, applicazione del *CSS*, ecc) ogni volta che si usa un semplice *tag* (ad esempio: *Grid, Button, TextEdit, CheckBox, FormLayout, ecc*), si è deciso di crearlo come un "componente di base", al quale è possibile comunque modificare o aggiungere funzionalità;

    · *Services*: include la cartella *Interfaces*, per le classi dichiaranti interfacce, e *Implementations*, per le classi che le implementano. I *services* corrispondono al *controller*.

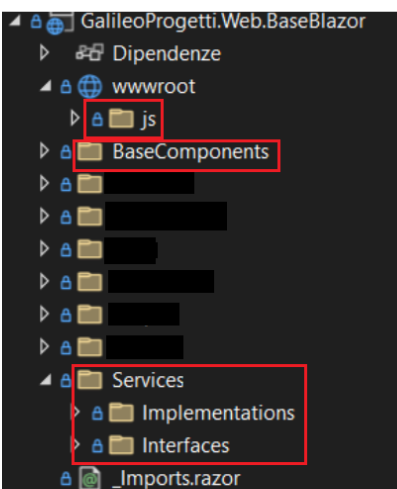


Figura 4.5: Suddivisone del progetto GalileoProgetti.Web.BaseBlazor

- *GalileoProgetti.Web.Blazor.Resources*: contiene i *file* di risorse *.resx* utilizzati per l'internazionalizzazione dell'applicazione. Il *file* per la traduzione italiana è nominato *nomefile.resx*, mentre quello per la lingua inglese *nomefile.en.resx*;

- *GalileoProgetti.Web.Blazor.Shared*: contiene la cartella *DTO*, che include le classi del modello, infatti le pagine prelevano i dati delle classi contenute in questa cartella;

- *GalileoProgetti.Web.BlazorTest*: contiene i *test* dell'applicazione;

- *GalileoProgetti.Web.MatchBlazor*: contiene una cartella *Pages*, la quale include l'implemetazione delle pagine dell'applicazione. Le *pages* corrispondono alla vista.

La figura 4.6 illustra i sei progetti appena spiegati in cui la soluzione *Blazor* è suddivisa. L'ultimo progetto appare in grassetto, in quanto è il progetto di avvio dell'applicazione.

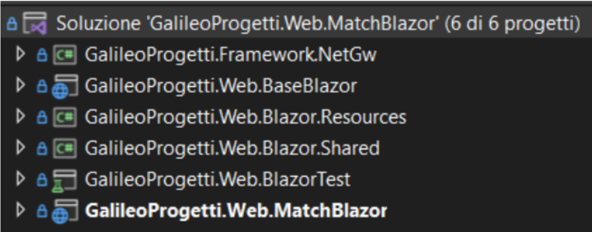


Figura 4.6: Organizzazione della soluzione GalileoProgetti.Web.MatchBlazor

4.3 Developer Express

Developer Express, abbreviato in *DevExpress*, è una società di sviluppo *software* con sede negli Stati Uniti. Produce strumenti e componenti di assistenza alla codifica per sviluppatori *Delphi, C++ Builder e Microsoft Visual Studio*. La maggior parte della sua linea di prodotti è costituita da componenti *VCL, .NET WinForms e ASP.NET* che replicano l'interfaccia utente delle applicazioni *Microsoft Windows e Microsoft Office*.
La *suite* di componenti dell'interfaccia utente di *Blazor DevExpress* viene fornita con una *suite* completa di componenti *Blazor* nativi (inclusi una *Grid*, un'utilità di pianificazione, un grafico, editor di dati, editor di testo e *report*). I controlli dell'interfaccia utente *DevExpress Blazor* vengono forniti come parte della sottoscrizione *DevExpress ASP.NET*. Per la compresione dell'utilizzo dei componenti si è consultato: *DevExpress Blazor* UI_G *Components*[4].

[4] *DevExpress Blazor*. URL: https://demos.devexpress.com/blazor/.

Capitolo 5

Sviluppo

5.1 Struttura di una pagina Blazor

In questo paragrafo viene spiegata la struttura generale di un *file*, in base alla sua estensione.

5.1.1 Componente Razor

Il componente è il *file* che contiene i tag *HTML* per l'implementazione della pagina web dell'applicazione. Il *file* è nominato in *nomefile.razor*. Ogni componente, in testa, prima dei *tag*, contiene le seguenti direttive:

- *@page "..."* per inserire il *path* per la navigazione;
- *@using ...* per l'utilizzo di eventuali *namespace*;
- *@inherits* per specificare un'eventuale classe base;
- *@attribute [Authorize]* per indicare che alla classe attribuita sono applicate le specifiche autorizzazioni;
- *@RenderBase* è una porzione di UI_G collegata al tipo *RenderTreeBuilder*.

Dopo l'implementazione con i *tag HTML*, è possibile scrivere la parte di codice *C#*, racchiundendola all'interno di questo blocco *@code{ ... }*.

5.1.2 Classe C#

Per la realizzazione dell'applicazione si è preferito inserire il codice *C#* in un *file* diverso, il quale viene tassativamente nominato *nomefile.razor.cs*, perchè in questo modo, al contrario della scrittura *nomefile.cs*, la classe viene associata automaticamente al componente.
La classe viene dichiarata, all'interno di un *namespace*, *partial*. Questa *keyword* indica che altre parti della classe, della struttura o dell'interfaccia possono essere definite nel *namespace*. Tutte le parti devono utilizzare la parola chiave *partial*, devono essere disponibili al momento della compilazione per formare il tipo finale e devono avere lo stesso livello di accessibilità (*public, private e protected*).
Ogni classe richiama il metodo *OnInitialized* o *OnInitializedAsync*.

OnInitializedAsync

OnInitializedAsync è un metodo che viene invocato quando il componente è pronto per l'avvio. Ogni classe associata al componente *razor* di una pagina, all'interno di questo metodo, contiene un blocco *try* che, tramite una variabile, salva il risultato dell'azione di *load* e di configuazione della pagina. Questa varibile viene poi utilizzata per la corretta associazione al modello e agli opportuni messaggi del *server*. Infine la varibile booleana che gestisce il *loading* della pagina viene impostata a *false*, in quanto le operazioni sono terminate. Il blocco *catch* include il tipo di eccezione e contiene istruzioni aggiuntive necessarie per gestirla. La figura 5.1 illustra il metodo appena descritto.

```
protected override async Task OnInitializedAsync()
{
    try
    {
        var result = await CommandService.ExecuteLoadAction<AnagraficaDto>
            (Id.ToString(), MatchPage).ConfigureAwait(false);
        Model = result.Model;
        MessageItems = result.ServerMessage;
        InLoading = false;
    }
    catch (Exception e)
    {
        ErrorComponent.ShowError(e);
    }

    await base.OnInitializedAsync();
}
```

Figura 5.1: Implementazione del metodo OnInitializedAsync

5.1.3 Applicazione CSS

Per l'applicazione dei fogli di stile *CSS*, analogamente, il file deve essere nominato *nomefile.razor.css*. Infatti in questo modo l'associazione con il *file .razor* è automatica.

5.2 Component Loader

Tutti i *tag* del componente *razor* di un *widget* sono racchiusi all'interno di un *Component Loader*, il quale è un "componente di base" che è stato creato per la gestione della visualizzazione del *loading*, se il *widget* sta caricando i dati, oppure della visualizzazione di un messaggio d'errore con opportune spiegazioni nel caso ci fossero stati problemi. Il componente è stato implementato (illustrato in figura 5.2) con un semplice *if*: i *div* contengono la parte di *loading*, mentre *ErrorContent e ChildContent* sono dei *RenderFragment*, cioè porzioni di UI_G.

```
@if (Visible)
{
    <div class="splash-new-screen" >
        <div class="spinner-border"></div>
        <div class="splash-screen-text"> Loading.</div>
    </div>
}
else
{
    @if (IsInError)  { @ErrorContent }
    else { @ChildContent }
}
```

Figura 5.2: Implementazione del componente ComponentLoader

Per l'utilizzo del componente (illustrato nella figura 5.3), è necessario utilizzare due varibile booleane, di cui una associata al valore *@bind-Visbile* e un'altra associata al valore *@bind-IsInError*.

```
<ComponentLoader @bind-Visible="@WidgetLoading" @bind-IsInError="@IsInError" >
```

Figura 5.3: Utilizzo di ComponentLoader

5.3 Implementazione dei widget

In questa sezione viene spiegato come sono stati implementati i *widget* mostrati in figura 5.4 della pagina *"Dati Generali"*. Tutti e tre i *widget* sono dei "componenti di base", infatti non vengono implementati all'interno di un componente che diventerà pagina, ma in un componente a sè.

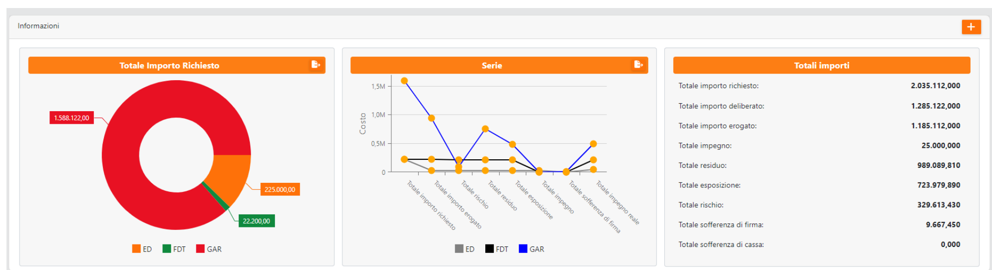


Figura 5.4: Visualizzazione widget della pagina Dati Generali

Per prima cosa è stata creata la classe *WidgetData*, che è una classe pubblica che contiene una coppia di valori per individuare un punto nel *chart*. La classe *WidgetDataSerie* include il titolo del *chart* e la lista, di tipo *List<WidgetData>*, contenente tutte le coppie di valori per realizzare il grafico. Nell'interfaccia *IWidgetDataService* sono dichiarati i metodi che verranno invocati dalle classi dei *widget*, mentre nella classe *WidgetDataService* questi metodi vengono implementati. Il metodo principale di quest'ultima classe è *GetDataWidget*, il quale come parametri richiede:

- *PageKindEnum*: è un enumeratore che contiene l'argomento generale della pagina (*Anagrafica, Pratica, Finanziamento*);
- *string*: corrisponde all'identificativo del *widget*, esempio *id1, id2 o id3*;

- *WidgetKindEnum*: è un enumeratore che contiene gli argomenti che un *widget* può mostrare (*Totale Importo Richiesto, Totale importo Erogato, Totale Esposizione, ecc.*)

Questo metodo, tramite uno *switch* sul parametro *WidgetKindEnum*, ritorna un oggetto *WidgetDataSerie* con l'opportuna lista e il titolo corretto.
Un caso particolare è quello del *PointSeriesChart*, in quanto un oggetto di tipo *WidgetData* non è sufficiente per raffigurare le tre serie. Quindi, si è creata una classe *WidgetDataPointSeries*, la quale contiene tre coppie di valori x ed y, ognuna valente per individuare un punto di una serie. Analogamente, la classe *WidgetDataSerie* contiene una lista con tutte le coppie di valori delle serie.

5.3.1 Widget Container

I tre componenti *widget e il Popup* sono contenuti all'interno di un altro "componente di base" chiamato *Widget Container*. I tre *widget* sono dichiarati nella classe *WidgetContainer.razor.cs* come *Render Fragment* e vengono valorizzati nel metodo *OnInitializedAsync*. Infatti, come mostrato in figura 5.5, viene eseguito il metodo *LoadDataAsync*, il quale si occupa di assegnare il giusto argomento ad ogni *widget* (questo metodo verrà approffondito nella sezione 5.3.5). Infine, per ognuno, viene invocato *Widget Factory*.

```
protected override async Task OnInitializedAsync()
{
    var loadData = await WidgetService.LoadDataAsync();
    Widget1 = WidgetFactory(loadData.Id1);
    Widget2 = WidgetFactory(loadData.Id2);
    Widget3 = WidgetFactory(loadData.Id3);
}
```

Figura 5.5: Implementazione del metodo OnInitializedAsync

Il metodo *WidgetFactory*, mostrato in figura 5.6, in base all'*id* che riceve in ingresso, crea un *Render Fragment* corrispondente ad uno dei tre *widget*.

```
public RenderFragment WidgetFactory(int id) => builder =>
{
    var data = WidgetService.GetDataManagement(id);
    builder.OpenComponent(0, (data.Type));
    builder.AddAttribute(1, "WidgetKind", data.EnumWidget);
    builder.CloseComponent();

};
```

Figura 5.6: Implementazione del metodo WidgetFactory

L'oggetto *DataManagement*, ritornato da *GetDataManagement(int)*, contiene:

- un intero per identificare il numero dell'importo;
- una stringa per definire la *caption*;
- una stringa per il nome completo dell'importo (esempio: *totale importo richiesto*);

- un *Type* per identificare il tipo del *widget* (esempio: *WidgetDonutChart*);
- *PageKindEnum* per l'argomento generale della pagina;
- *WidgetKindEnum* per l'argomento del *widget*.

Il metodo *OpenComponent(int, Type)* aggiunge un *frame* che rappresenta un componente figlio, *AddAttribute(int, string, object)* aggiunge un *frame* che rappresenta un attributo con valori stringa e *CloseComponent* contrassegna il *frame* del componente aggiunto in precedenza come chiuso.

5.3.2 Donut Chart

DevExpress ha un componente *DxPieChart<T>* per realizzare un grafico a torta di tipo *T*. Per prima cosa è necessario inserire le seguenti informazioni:

- Il riferimento, tramite la direttiva *@ref*, al proprio grafico;
- La fonte di dati, ovvero inserire la lista contenente tutti i valori per creare il *chart*;
- Il tipo del parametro *T* del proprio *chart*.

A tale *chart* deve essere collegata una serie per il corretto prelievo dei dati. La serie *DxPieChartSeries* contiene due campi, *ValueField e ArgumentField*, che tramite un'*espressione lamba* permette di specificare l'oggetto della classe *T* che deve essere considerato. All'interno di questo *tag* devono sempre essere specificati i tipi del *chart*, dell'argomento (*TArgument*) e del valore (*TField*).
In figura 5.7 viene mostrato l'implementazione di quanto appiena spiegato.

```
<DxPieChart @ref="@chart" Data="@Data" Diameter="diameter"
            InnerDiameter="innerDiameter" T="WidgetData">
    <DxPieChartSeries T="WidgetData" TArgument="string" TValue="double"
        ValueField="(si => (double)si.Xvalue)" ArgumentField="@(si => (string)si.Yvalue)"
        SummaryMethod="Enumerable.Sum" >
```

Figura 5.7: Implementazione parziale del componente WidgetDonutChart

Implementazione della classe WidgetDonutChart

La classe *WidgetDonutChart* esegue l'*inject* dell'interfaccia *IWidgetDataService* come spiegato nella sezione 4.1.2 ed implementa il metodo *OnInitialized* nel seguente modo:

- Il *loading* del *widget* viene attivato. L'utente visualizzerà il *widget* come mostrato in figura 5.8;

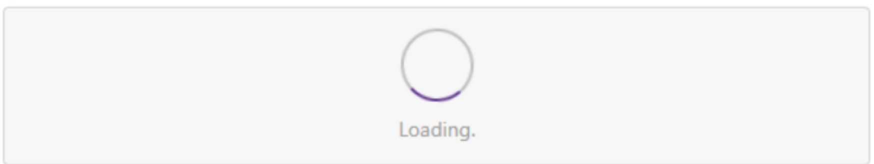


Figura 5.8: Visualizzazione del loading di un widget

- Viene invocato il metodo *GetDataWidget* (spiegato nella sezione 5.3), salvando il contenuto in una variabile;
- La lista contenuta nella variabile viene assegnata alla lista *Data*;
- Il titolo contenuto nella varibile viene assegnato alla stringa Titolo;
- Infine, viene disattivato il *loading*.

Nella figura 5.9 viene mostrata l'implementazione del metodo appena descritto.

```
protected override void OnInitialized()
{
    try
    {
        WidgetLoading = true;
        var widgetResponse = WidgetService.GetDataWidget(PageKind, "id1", WidgetKind);
        Data = widgetResponse.DataSerie;
        Title = widgetResponse.Title;
        WidgetLoading = false;
    }
    catch (Exception e)
    {
        IsInError = true;
        WidgetLoading = false;
        exception = e;
        _smallMessageDetails = exception.ToString() + Environment.NewLine + exception.StackTrace;
    }
}
```

Figura 5.9: Implementazione del metodo OnInitialized nella classe WidgetDonutChart

Per esportare il *donut chart* è presente un metodo *Export* implementato come illustrato nella figura 5.10. Il metodo *TryParse<T>(string, bool, out T)* converte la rappresentazione di stringa del nome o del valore numerico di una o più costanti enumerate in un oggetto enumerato equivalente. Il parametro *bool* specifica se l'operazione verrà eseguita con distinzione tra maiuscole e minuscole (*case-sensitive*). Il valore restituito, rappresentato da un booleano, indica se la conversione è riuscita. Infine viene poi esportato con le caratteristiche scelte.

```
public void Export()
{
    ChartExportFormat format = ChartExportFormat.Png;
    Color backgroundColor = Color.White;
    int margin = 4;
    Time = DateTime.Now;
    if (Enum.TryParse<ChartExportFormat>("Png", true, out format))
        chart?.ExportAsync(Title + "_" + Time, format, margin, backgroundColor);
}
```

Figura 5.10: Implementazione del metodo Export

Implementazioni facoltative per la realizzazione del donut chart

Nel componente *razor* sono state aggiunte informazioni per realizzare il *donut chart* in modo più funzionale e originale. Per trasformarlo in un grafico "a ciambella" si è inserita una misura per il diametro interno del cerchio. Si è impostato, poi, che la *label* del *chart* fosse visibile, esterna al grafico e in formato *fixed point*. Si è deciso di rendere visibile anche la legenda per facilitare la comprensione del *chart*.
In figura 5.11 viene illustrata la realizzazione finale del *widget* contenente il *donut*

chart. In particolare, se il cursore viene messo sopra ad una delle porzioni del grafico, compare un piccolo riquadro con le informazioni riguardo a quella parte.

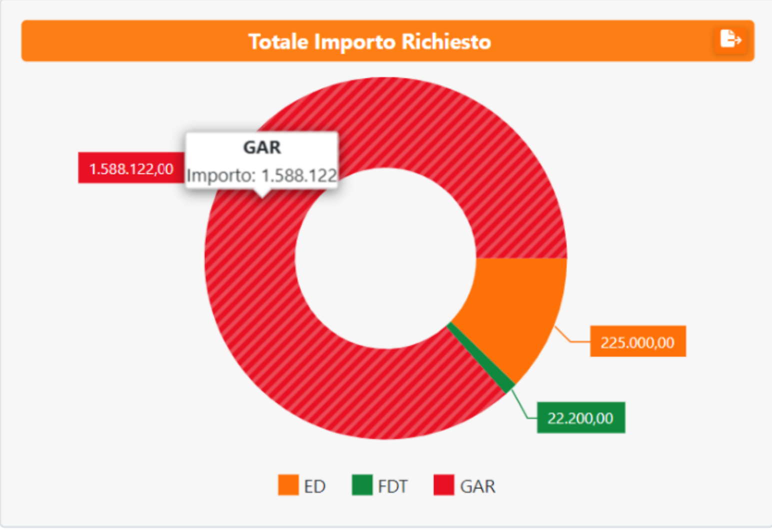


Figura 5.11: Visualizzazione del widget contenente il donut chart

Potrebbe capitare che durante le operazioni di caricamento dei dati del *widget* sorgano problemi e quindi l'utente visulizzerà un messaggio come mostrato in figura 5.12.

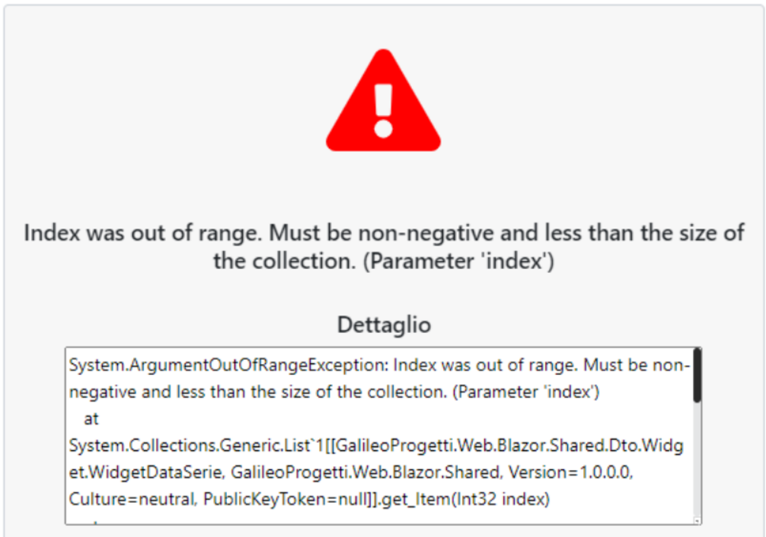


Figura 5.12: Visualizzazione errore di un widget

5.3.3 Point Series Chart

DevExpress ha un componente *DxChart<T>* per realizzare un grafico di tipo *T*. Per prima cosa è necessario inserire le seguenti informazioni:

- Il riferimento, tramite la direttiva *@ref*, al proprio grafico;

- La fonte di dati, ovvero inserire la lista contenente tutti i valori per creare il *chart*;

- Il tipo del parametro *T* del proprio *chart*.

A tale *chart* deve essere collegata una serie, per il corretto prelievo dei dati. La serie *DxChartLineSeries* contiene due campi, *ValueField e ArgumentField*, che tramite un'*espressione lamba* permette di specificare l'oggetto della classe *T* che deve essere considerato. All'interno di questo *tag* devono sempre essere specificati i tipi del *chart*, dell'argomento (*TArgument*), del valore (*TField*) e il nome della serie.
In figura 5.13 viene mostrato l'implementazione di quanto appiena spiegato.

```
<DxChart @ref="@chart" Data="@Serie" Width="100%" T="WidgetDataPointSeries">
    <DxChartLineSeries Color="@Color.Gray" T="WidgetDataPointSeries" TArgument="string" TValue="double"
                       ValueField="@( i => (double)i.Yvalue)" ArgumentField="@(i => (string)i.Xvalue)"
                        Name="ED">
```

Figura 5.13: Implementazione parziale del componente WidgetPointSeriesChart

Implementazione della classe WidgetPointSeriesChart

La classe *WidgetPointChart* esegue l'*inject* dell'interfaccia *IWidgetDataService* come spiegato nella sezione 4.1.2. Il metodo *OnInitialized* e *Export* sono stati implementati in modo analogo a quanto riportato nella sezione 5.3.2.

Implementazioni facoltative per la realizzazione del point series chart

Nel componente *razor* sono state aggiunte informazioni per realizzare il *point series chart* in modo più funzionale e originale. All'interno di ogni *DxChartLineSeries*, tramite il *tag DxChartSeriesPoint*, si è scelto il simbolo dei punti, la dimensione e il colore, in modo da personalizzarli e differenziarli per ogni serie. Anche in questo caso la legenda aiutava alla comprensione del grafico e quindi è stato scelto di renderla visibile. Si è deciso il formato in cui visualizzare gli importi e si è potuto inserire un titolo per l'asse delle ordinate.
In figura 5.14 viene illustrata la realizzazione del *widget* contenente il *point series chart*. In particolare, se il cursore viene messo sopra un punto del grafico, compare un piccolo riquadro con le informazioni riguardo a quella serie e la linea aumenta di spessore.

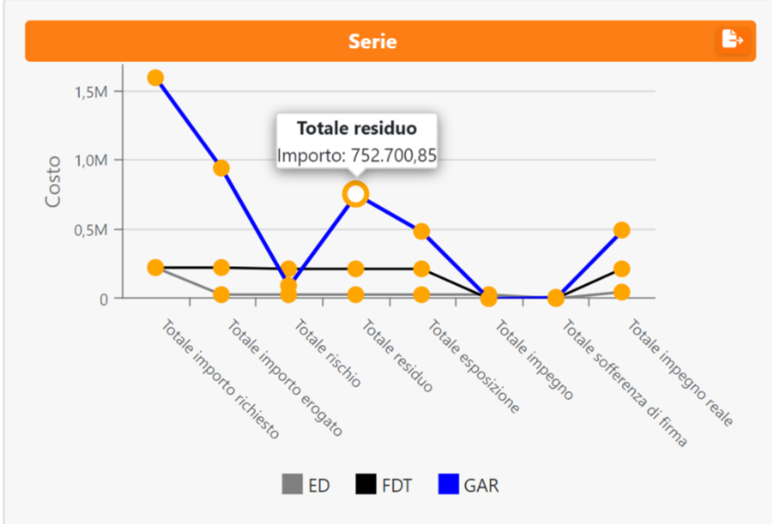


Figura 5.14: Visualizzazione del widget contenente il point series chart

5.3.4 Elenco Importi Totali

Il *widget* contenente l'elenco degli importi totali è stato implementato con dei basilari *tag HTML*. Semplicemente si itera la lista contenente tutti gli importi e per ognuno viene riportato il tipo (*totale esposizione, totale importo richiesto, ecc*) e il valore.
In figura 5.15 viene mostrata l'implementazione all'interno del *Child Content* del *widget* riguardante gli importi totali.

```
<ChildContent>
    <p class="title-one-info-widget"> @InfoResponse.Title </p>
    @foreach (var oggetto in Data)
    {
        <p class="descrizione-valore-misura"> @oggetto.Xvalue:
            <strong class="valore-right">  @oggetto.YvalueType   </strong>
        </p>
    }
</ChildContent>
```

Figura 5.15: Implementazione parziale di OneInfoWidget.razor

Implementazione della classe OneInfoWidget

La classe *OneInfoWidget* esegue l'*inject* dell'interfaccia *IWidgetDataService* come spiegato nella sezione 4.1.2. Il metodo *OnInitialized* è stato implementato in modo analogo a quanto riportato nella sezione 5.3.2. In aggiunta però, prima della disattivazione del *loading*, viene eseguito un ciclo *for*, il quale sovrascrive il valore dell'importo in un formato specifico, ad esempio *10000* diventa *10.000,00* (gli importi rigorosamente non devono avere il simbolo dell'euro (€)).
È presente nuovamente il metodo *TryParse(string, out double)*, ma questa volta con parametri in ingresso differenti. Tale metodo converte la rappresentazione di stringa di un numero nel rispettivo numero a virgola mobile a precisione doppia equivalente. Restituisce un valore booleano per indicare se la conversione è riuscita o meno. Viene utilizzato anche un altro metodo, *ToString(string, IFormatProvider)* che formatta il

valore dell'istanza corrente usando il formato specificato.
In figura 5.16 viene mostrato il *widget* con l'elenco degli importi. Per ognuno di essi, è possibile visualizzare il corrispettivo *donut chart*, suddiviso in base al tipo di finanziamento.

Totali importi	
Totale importo richiesto:	2.035.112,000
Totale importo deliberato:	1.285.122,000
Totale importo erogato:	1.185.112,000
Totale impegno:	25.000,000
Totale residuo:	989.089,810
Totale esposizione:	723.979,890
Totale rischio:	329.613,430
Totale sofferenza di firma:	9.667,450
Totale sofferenza di cassa:	0,000

Figura 5.16: Visualizzazione del widget con l'elenco degli importi totali

5.3.5 Popup Scelta Widget

Il *Popup* realizzato è mostrato in figura 5.17.

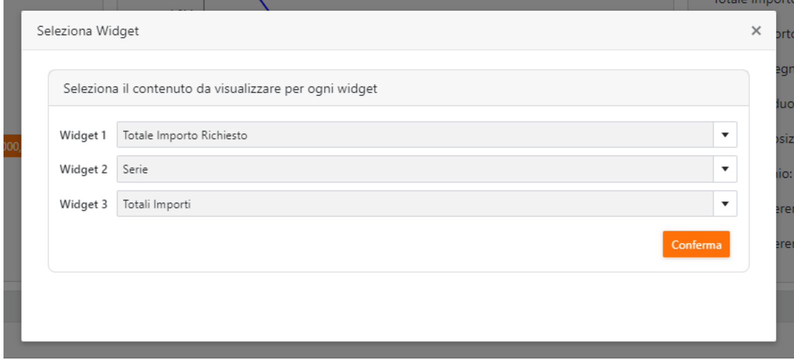


Figura 5.17: Visualizzazione della scelta di ogni widget

È stato creato all'interno della classe *WidgetContainer*, tramite il metodo *NewPopUp()*. La classe effettua l'*inject* dell'interfaccia *IPopUpServices*, la quale contiene il metodo *CreatePopUp*. Quest'ultimo viene invocato da *NewPopUp()* come mostrato in figura 5.18.

```
protected void NewPopUp()
{
    PopupInfoWidget = ServicesPopup.CreatePopUp(this, OnPopClosing, "Seleziona Widget", "800", "350",
                            () => new DynamicComponentFactory<WidgetChooser>()
                           .AddPropertyValue(p => p.PageKindEnum, this.PageKind));
    StateHasChanged();
}
```

Figura 5.18: Implementazione del metodo NewPopUp

Infatti, per ottenere un *RenderFragment*, bisogna invocare *CreatePopUp* con i seguenti parametri:

- *IHandleEvent* che in questo caso è semplicemente *this*;
- L'*Action* di chiusura del *popup*;
- Una stringa per il titolo;
- Una stringa per la larghezza e una per l'altezza;
- Una funzione <*IDynamicComponentFactory*<*T*>.

Viene poi impostata la pagina presso cui si trova.
Una volta aperto il *popup* dall'applicazione, ci sarà la possibilità di scegliere cosa visualizzare in ogni *widget* tramite un menù a tendina e ogni voce è associata ad un'*id*. (Questa schermata è un altro "componente di base" chiamato *Widget Chooser*.)
Successivamente, quando si premerà il pulsante *Conferma*, verrà invocato il metodo *Confirm()*, il quale salva in un oggetto gli *id* di ogni scelta e poi lo *serializza*$_G$ (vedesi figura 5.19) ottenendo una stringa con le informazioni. Infine, chiama un metodo *Javascript* per salvare le scelte dell'utente definitivamente (fino al suo prossimo cambiamento), in modo che ogni volta che aprirà l'applicazione vedrà sempre i *widget* scelti.

```
var dataJson = System.Text.Json.JsonSerializer.Serialize(combo);
js.SetInLocalStorage("AnagraficaData", dataJson);
```

Figura 5.19: Serializzazione dell'oggetto combo contenente gli id dei widget

Come accennato nella sezione 5.3.1, il metodo *LoadDataAsync* assegna il giusto argomento ad ogni *widget*. Ma, prima di fare questo, deve controllare se ha già dei valori in memoria (altrimenti assegnerà quelli di *default*): viene invocato un metodo *Javascript* che ritorna una stringa con le informazioni pertinenti se in memoria sono state salvate le scelte, altrimenti ritorna una stringa vuota. Successivamente avviene il processo inverso a prima, ovvero la *deserializzazione*$_G$ della stringa in modo tale da ottenere l'oggetto (vedesi figura 5.20).

```
var dataJson = await js.GetFromLocalStorage<string>("AnagraficaData");
if (!string.IsNullOrEmpty(dataJson))
{
    return System.Text.Json.JsonSerializer.Deserialize<WidgetChooser>(dataJson, defaultJsonSerializerOptions);
}
```

Figura 5.20: Serializzazione della stringa dataJson

Nelle figure 5.21 e 5.22 vengono riportati degli esempi sulle possibili scelte di visualizzazione dei *widget*.
In questo caso vengono mostrati tre *donut chart*.

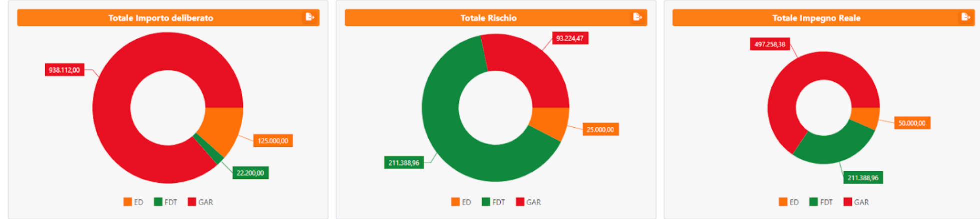


Figura 5.21: Visualizzazione di tre donut chart

In questo caso, invece, viene mostrato l'elenco degli importi totali, un *donut chart* e il *point series chart*.

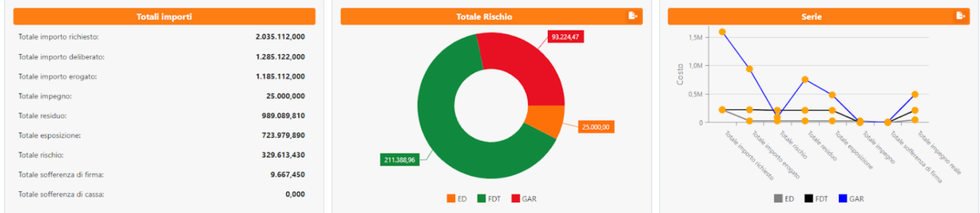


Figura 5.22: Visualizzazione elenco importo totali, donut chart e point series chart

5.4 Sviluppo di un form

Il contenuto di un *form* viene inserito all'interno di *<EditForm>* impostando il riferimento ad esso, al modello e inserendo un contesto. Successivamente viene creato un *DxFormLayoutGroup*, ovvero il contenitore di tutti i *DxFormLayoutItem*. A sua volta, ognuno di quest'ultimo, contiene un componente del *form*, ad esempio: *button, textedit, checkbox, ecc.* La maggior parte dei semplici *tag HTML* sono stati fatti diventare dei "componenti di base". Ognuno per prelevare i dati dal modello utilizza *@bind-Value*. I *tag* che si sono utilizzati sono i seguenti:

- *GalDateEdit*: implementa *DxDateEdit* e consente di modificare i valori delle date tramite un calendario a discesa integrato. Si possono inserire i valori della data direttamente nella casella di testo o selezionarne uno dal menù a discesa. Viene mostrato un esempio di utilizzo in figura 5.23;

```
<DxFormLayoutItem Caption="@ResxForm.DateCessazione" ColSpanSm="3">
    <GalDateEdit Id="txtDataCessazione" @bind-Value="@Model.DatiImpresa.Cessazione"/>
</DxFormLayoutItem>
```

Figura 5.23: Esempio di utilizzo di GalDateEdit

- *GalCheckBox*: implementa *DxCheckBox* e consente agli utenti di selezionare sì/no o vero/falso. Oltre al collegamento con il modello, si possono aggiungere particolarità, come nel caso in figura 5.24, indicando la *label* e la sua posizione;

```
<GalCheckBox LabelPosition="LabelPosition.Right"
        Label="@ResxForm.Associata" @bind-Value="Model.DatiImpresa.Associata"  />
```

Figura 5.24: Esempio di utilizzo di GalCheckBox

- *GalTextEdit*: implementa *DxTextBox* ed è la casella di testo che consente ad un utente di immettere e modificare una singola riga di esso. Viene mostrato un esempio di utilizzo in figura 5.25;

```
<GalTextEdit Id="txtRagioneSociale" @bind-Value="@Model.RagioneSociale" />
```

Figura 5.25: Esempio di utilizzo di GalTextEdit

- *GalLookUp*: implementa *DxTextBox* ed è la casella di testo con il valore in sola lettura. Viene mostrato un esempio di utilizzo in figura 5.26;

```
<GalLookup Id="txtCodiceVAT" @bind-Value="@Model.IdNazioneVAT" />
```

Figura 5.26: Esempio di utilizzo di GalLookUp

- *GalNumericEdit*: implementa *DxSpinBox* e consente di visualizzare e modificare valori numerici. Viene mostrato un esempio di utilizzo in figura 5.27;

```
<GalNumericEdit Id="txtIdAzienda" @bind-Value="@Model.IdAzienda" />
```

Figura 5.27: Esempio di utilizzo di GalNumericEdit

- *DxComboBox*: è il componente di *DevExpress* e consente agli utenti di selezionare un elemento da un elenco a discesa tramite un *clic*. Per questo componente è necessario inserire una lista (*Data*) che contenga tutte le voci delle possibili scelte, il *TextFieldName* e il *ValueFieldName*. Viene mostrato un esempio di utilizzo in figura 5.28;

```
<DxComboBox Id="dropIdConfidiSedeModifica" Data="@DatiGestoreDto.ListaModificaIns"
  TextFieldName="@nameof(DatiGestoreDto.Description)"
  ValueFieldName="@nameof(DatiGestoreDto.Id)" @bind-Value="@DatiGestoreDto.IdModIns"/>
```

Figura 5.28: Esempio di utilizzo di DxComboBox

5.5 Sviluppo di una griglia

DevExpress ha a disposizione un componente *DxGrid* che consente di visualizzare, gestire e modellare dati tabulari. Infatti, questo tipo di griglia permette facilmente di modificare le righe all'interno di essa. Per conoscere il corretto formato di ogni colonna (*string, int, int?, datetime, datetime?*, ecc) si è utilizzato *SQL Server Management Studio*, controllando le colonne del *database* collegato a *Match!*. Le impostazioni predefinite di una generica griglia sono state inserite all'interno di un componente chiamato *GalGridNew* strutturato nel seguente modo.
Tutta l'implementazione è racchiusa all'interno del *Component Loader*. All'interno del *tag DxGrid* sono stati inseriti:

- La direttiva *@ref* per il riferimento alla griglia;
- *KeyFieldName* ottiene o imposta il nome del campo chiave dell'origine dati;
- *Data* contiene la lista con i dati;

- *CssClass* per il riferimento alla classe *CSS*;
- *EditMode* specifica in che modo l'utente modifica la griglia;
- *EditFormTemplate* specifica il *template* usato per visualizzare la schermata di modifica.

Opzionali per la compilazione, ma necessari per lo sviuppo dell'applicazione sono stati aggiunti anche questi campi:

- *EditFormButtonVisbile* nasconde o meno la visualizzazione dei bottoni predefeniti (Annulla e Salva) del componente durante la modifica di una riga della tabella;
- *PageSize* specifica il massimo numero di righe che si vogliono visualizzare per pagina;
- *PageVisibleNumericButtonCount* specifica il massimo numero di bottoni, indicanti le pagine;
- *PageNavigationMode* specifica in che modo l'utente naviga da una pagina all'altra;
- *ShowGroupPanel* specifica se visualizzare in pannelli di gruppo;
- *AutoExpandAllGroupRows* specifica se espandere automaticamente tutti i gruppi di righe;
- *CustomGroup* consente di implementare la logica personalizzata utilizzata per raggruppare i dati;
- *CustomizeGroupValueDisplayText* consente di personalizzare il testo visualizzato all'interno di un gruppo.

Successivamente vengono inserite le colonne e una particolare è la *DxGridCommandColumn*, ovvero la colonna di comando. Infatti questa contiene di *default* dei buttoni per le operazioni di *new, change e delete* della riga. Per scelta, si è decisi di renderli non visibili, in quanto sono stati implementati nuovamente in modo da poterli rendere originali con delle icone. Per richiamare il metodo di *default* per ogni operazioni, si è fatta un'espressione *lambda* che invocasse sulla griglia il metodo *StartEditNewRowAsync()* per aggiungere una nuova riga, *StartEditRowAsync(int)* per modificare una riga e *ShowRowDeleteConfirmation(int)* per mostrare il dialogo di conferma eliminazione della riga. La figura 5.29 mostra il componente *GalGridNew*.

```
<ComponentLoader @bind-Visible="@GridLoading" >
  <ChildContent>
    @if (EnableEditMode)
        {
        <p class="container-p">
            <GalButton IconCssClass="fa-icon plus-small" CssClass="button-new" OnClick="(() => TheGrid.StartEditNewRowAsync())" Text="@InsertName" />
        </p>
        }
    <DxGrid @ref="@TheGrid" KeyFieldName="@KeyFieldName" Data="@Data"
            CssClass="@CssClass" EditMode="@EditMode" EditFormTemplate="@EditFormTemplate"
            EditFormButtonsVisible="false" PageSize="@PageSize" PagerVisibleNumericButtonCount="@PagerVisibleNumericButtonCount"
            PagerNavigationMode="PagerNavigationMode.NumericButtons"
            ShowGroupPanel="@ShowGroupPanel" AutoExpandAllGroupRows="@AutoExpandAllGroupRows"
            CustomGroup="@CustomGroup" CustomizeGroupValueDisplayText="@CustomizeGroupValueDisplayText" >
      <Columns>
        <DxGridCommandColumn NewButtonVisible="false" EditButtonVisible="false"  DeleteButtonVisible="false" Width="100px" Visible="@EnableEditMode" >
          <CellDisplayTemplate Context="CommandColumnTemplate2" >
            <div class="commandcolumn-buttons">
               <GalButton IconCssClass="fa-icon pen-solid" CssClass="btn-big-size"
                          OnClick="(() => TheGrid.StartEditRowAsync(CommandColumnTemplate2.VisibleIndex))" />
               <GalButton IconCssClass="fa-icon trash-small" CssClass="btn-big-size"
                          OnClick="(() => TheGrid.ShowRowDeleteConfirmation(CommandColumnTemplate2.VisibleIndex))" />
            </div>
          </CellDisplayTemplate>
        </DxGridCommandColumn>
        @Columns
      </Columns>
    </DxGrid>
  </ChildContent>
</ComponentLoader>
```

Figura 5.29: Implementazione del componente GalGridNew

5.5.1 Griglia con le operazioni di new, change e delete

Prendendo in considerazione la pagina *NoteLibroSociale.razor* possiamo notare dalla figura 5.30 che le informazioni della griglia sono molto meno rispetto a quelle delle *GalGridNew*, questo perchè alcuni valori erano già stati impostati e non serviva cambiarli. In particolare notiamo che per ottenere *Data*, preleviamo la lista specifica dal *Dto* associato alla classe. Le colonne, seppur anche queste diventate "componenti di base", *GalGridColumn*, contengono semplicemente il *FieldName* per specificare il campo data che fornisce i valori delle colonne e la *Caption* della colonna. (Nella sezione 5.6 verrà analizzata la scrittura all'interno della *caption*). Infine, per le griglie come questa che necessitano di essere modificabili, bisogna introdurre il *tag EditFormTemplate* per gestire tale parte. Si tratta di un banale *form*, con le stesse informazioni del paragrafo 5.4.

```
<GalGridNew @ref="grid" Data="@LibroSocialeDto.DataList"
            KeyFieldName="@nameof(LibroSocialeDto.IdGrid)" EnableEditMode="true">
    <Columns>
        <GalGridColumn FieldName="@nameof(LibroSocialeDto.Descrizione)" Caption="@ResxGridColumn.Descrizione" Width="100%" />
        <GalGridColumn FieldName="@nameof(LibroSocialeDto.DescrizioneEstesa)" Caption="@ResxGridColumn.DescrizioneEstesa" Width="100%" />
        <GalGridColumn FieldName="@nameof(LibroSocialeDto.LibroSociale)" Caption="@ResxGridColumn.LibroSociale" Width="100%" />
        <GalGridColumn FieldName="@nameof(LibroSocialeDto.DataInserimento)" Caption="@ResxGridColumn.DataInserimento" Width="100%" />
        <GalGridColumn FieldName="@nameof(LibroSocialeDto.DataStampa)" Caption="@ResxGridColumn.DataStampa" Width="100%" />
        <GalGridColumn FieldName="@nameof(LibroSocialeDto.TipoDelibera)" Caption="@ResxGridColumn.TipoDelibera" Width="100%" />
        <GalGridColumn FieldName="@nameof(LibroSocialeDto.DataDelibera)" Caption="@ResxGridColumn.DataDelibera" Width="100%" />
    </Columns>
    <EditFormTemplate Context="editFormContext">
            @{
                var NoteLibroSociale = (NoteLibroSocialeDto) editFormContext.EditModel;
            }
            <DxFormLayout  CssClass="w-100">
            <DxFormLayoutItem Context="EditForm2" Caption="@ResxGridColumn.LibroSociale" ColSpanSm="3">
            <DxComboBox Id="dropIdLibroSociale" Data="@LibroSocialeDto.ComboList" TextFieldName="@nameof(LibroSocialeDto.Description)"
                        ValueFieldName="@nameof(LibroSocialeDto.Id)" @bind-Value="@NoteLibroSociale.IdLibroSociale" />
            </DxFormLayoutItem>
            <DxFormLayoutItem Context="EditForm2" Caption="@ResxGridColumn.DataInserimento" ColSpanSm="2">
                <GalDateEdit Id="txtDataInserimento" @bind-Value="@NoteLibroSociale.DataInserimentoForm" />
            </DxFormLayoutItem>
```

Figura 5.30: Implementazione parziale del componente NoteLibroSociale

Nella figura 5.31 viene mostrata come appare una riga della griglia.

Figura 5.31: Visualizzazione di una riga della griglia modificabile

Nella figura 5.32 viene mostrato il *form* per inserire una nuova riga. Simile è il caso in cui si volesse modificare, infatti l'unica differenza è che in quest'ultima i valori presenti nella tabella compaiono anche nel *form*.

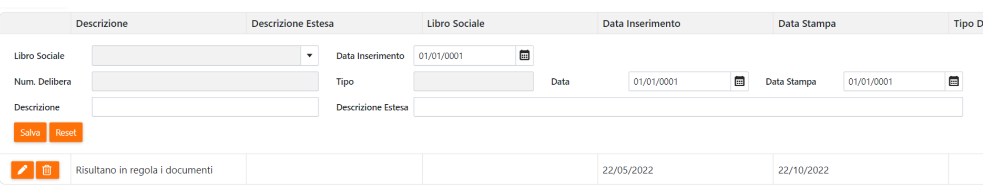


Figura 5.32: Visualizzazione del form per inserire una nuova riga

5.5.2 Griglia con raggruppamenti

Prendendo ora in considerazione la pagina *StrumentiFinanziari.razor* possiamo notare dalla figura 5.35 che i valori riguardo i raggruppamenti sono impostati a *true* e che sono stati implementati due nuovi metodi:

- *Grid_CustomGroup()*, illustrato in figura 5.33, raggruppa gli oggetti che hanno *FieldName* uguale a "Tipo";

```
void Grid_CustomGroup(GridCustomGroupEventArgs e)
{
    if (e.FieldName == "Tipo")
    {
        var str1 = Convert.ToString(e.Value1);
        var str2 = Convert.ToString(e.Value2);

        e.SameGroup = str1.Equals(str2);
        e.Handled = true;
    }
}
```

Figura 5.33: Implementazione del metodo Grid_CustomGroup()

- *Grid_CustomizeGroupValueDisplayText()*, illustrato in figura 5.34, si occupa della visualizzazione del testo per ogni raggruppamento.

```
void Grid_CustomizeGroupValueDisplayText(GridCustomizeGroupValueDisplayTextEventArgs e)
{
    if (e.FieldName == "Tipo")
    {
        string val = Convert.ToString(e.Value);
        string displayText = string.Format(val);
        e.DisplayText = displayText;
    }
}
```

Figura 5.34: Implementazione del metodo Grid_CustomizeGroupValueDisplayText()

Un'altra differenza è che la prima colonna è implementata diversamente: infatti contiene *DisplayFormat* per specificare il formato della stringa e *GroupIndex* per specificare l'indice della colonna tra le colonne raggruppate.

```
<GalGridNew @ref="grid" Data="@strumentiFinanziariDto.GetListStrumentiFinanziari()" PageSize="10"
            ShowGroupPanel="true" AutoExpandAllGroupRows="true"
            CustomGroup="@Grid_CustomGroup" CustomizeGroupValueDisplayText="@Grid_CustomizeGroupValueDisplayText" >
    <Columns>
        <GalGridColumn DisplayFormat="c" FieldName="@nameof(StrumentiFinanziariDto.Tipo)" GroupIndex="0" Width="10%" />
        <GalGridColumn FieldName="@nameof(StrumentiFinanziariDto.Operazione)" Caption="@ResxGridColumn.Operazione" Width="150px" />
        <GalGridColumn FieldName="@nameof(StrumentiFinanziariDto.Sottoscrizione)" Caption="@ResxGridColumn.Sottoscrizione" Width="150px" />
        <GalGridColumn FieldName="@nameof(StrumentiFinanziariDto.Delibera)" Caption="@ResxGridColumn.Delibera" Width="150px" />
```

Figura 5.35: Implementazione del componente Strumenti Finanziari

In figura 5.36 viene riportata la griglia appena descritta.

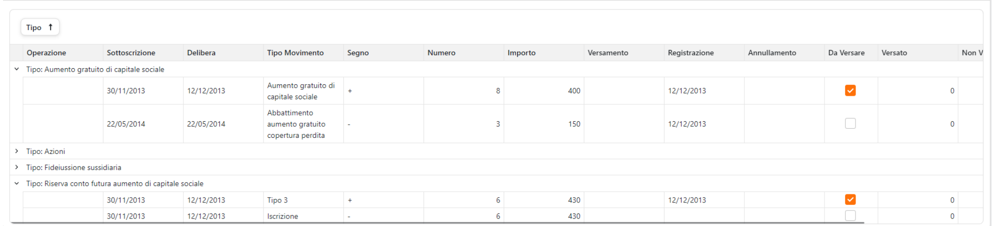


Figura 5.36: Visualizzazione della griglia della pagina Strumenti Finanziari

5.6 Internazionalizzazione

L'internazionalizzazione delle pagine dell'applicazione, ovvero la loro traduzione in lingua italiana o inglese, è stata fatta in due modi: inizialmente tramite l'uso della dipendenza da *IStringLocalizer<T>* e successivamente tramite accesso diretto al *file* contenente le traduzioni. Infatti, alla base di entrambi i modi, c'è l'utilizzo di un *file di risorse .resx*, che appare come illustrato in figura 5.37. Ad ogni nome viene associato un valore, che è quello che verrà mostrato a schermo sulla pagina.

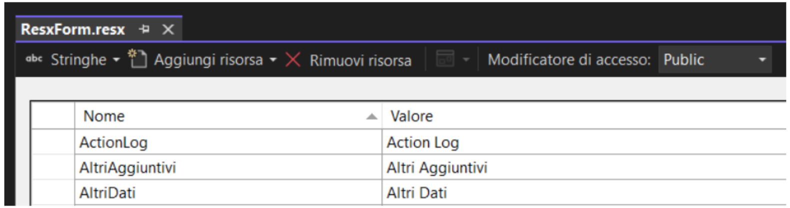


Figura 5.37: Struttura di un file di risorse

5.6.1 Utilizzo di IStreamLocalizer<T>

Per utilizzare *IStringLocalizer<T>* è necessario configurare il servizio nel *file Program.cs* tramite la sintassi *services.AddLocalization()*. Successivamente bisogna iniettare l'interfaccia nel componente che ne prevede l'uso (come spiegato nel paragrafo 4.1.2) e il tipo *T* corrisponderà al nome del *file .resx* che contiene le parole internazionalizzate. Infine per ogni *caption*, si applica la traduzione nel modo illustrato in figura 5.38. (*localizer* è il nome dell'oggetto su cui è stato fatto *@inject*).

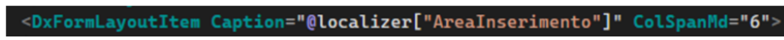


Figura 5.38: Utilizzo di IStringLocalizer<T>

Questo metodo però presenta un problema grave: se la stringa inserita tra gli apici non corrisponde a nessun nome del *file .resx*, il compilatore non segna errore, e quando navigheremo nell'applicazione ci accorgeremo che quella parola non è stata tradotta. Ovviamente questa non è una buona procedura, perchè si rischia di lasciare parole nell'applicazione non tradotte (in questo caso se la stringa fosse sbagliata visualizzeremmo *AreaInserimento* sia in italiano sia in inglese).

5.6.2 Accesso diretto al file di risorse

Per evitare questo problema è possibile accedere direttamente al *file di risorse*, che è anche più veloce. Infatti basta semplicemente utilizzare la sintassi rappresentata in figura 5.39. In questo modo, se si scrive ad esempio *Areainserimento* con la *i* minuscola o se si sbaglia grammaticamente a scrivere la parola, il compilatore segnerà subito errore, non trovandola nel *file .resx*.

```
<DxFormLayoutItem Caption="@ResxForm.AreaInserimento" ColSpanMd="6">
```

Figura 5.39: Esempio di accesso diretto al file di risorse

Capitolo 6

Verifica e validazione

6.1 Analisi statica

L'*analisi statica* è una tecnica di *testing del software* ed è il processo di valutazione di un sistema o di un suo componente basato sulla sua forma, sulla sua struttura, sul suo contenuto o sulla documentazione di riferimento. Ciò significa che la valutazione avviene a prescindere dall'esecuzione del sistema o dell'oggetto che si sta testando. L'*analisi statica* è stato il primo controllo effettuato sul codice e sui requisiti applicativi con l'intento di riscontrare anomalie nel prodotto finale.
I compilatori eseguono un'analisi statica per verificare che un programma soddisfi particolari caratteristiche di correttezza per poter generare il codice. Le anomalie rilevate dal compilatore all'interno del codice possono essere: nomi di identificatori non dichiarati, incoerenza tra tipi di dati coinvolti in un'istruzione e codice non raggiungibile dal flusso di controllo. In particolare si sono svolte le seguenti attività:

- *Code reading*: ovvero un'attenta e ripetitiva lettura del codice per controllare l'assenza di errori (di programmazione, di sintassi, ecc), *loop infiniti*, inefficienza di algoritmi e non strutturazione del codice;
- *Code refactoring*: ovvero la ristrutturazione del codice volta a renderlo più pulito e facile da manutenere. Con questa attività, è possibile preparare la struttura allo sviluppo di nuove *feature*, migliorandone le prestazioni.

6.2 Analisi dinamica

L'*analisi dinamica* è il processo di valutazione di un sistema *software* o di un suo componente basato sull'osservazione del suo comportamento in esecuzione. La precondizione necessaria per poter effettuare un *test* è la conoscenza del comportamento atteso per poterlo confrontare con quello osservato. Per una corretta *analisi dinamica* è necessaria un'adeguata analisi dei requisiti, la quale contiene tutte le caratteristiche grafiche e di usabilità desiderate. Si sono testati tutti gli *Use Case*$_G$ della sezione 3.2.

6.3 Test di accessibilità

I *test di accessibilità* sono un tipo di *test* di usabilità e sono utilizzati principalmente per rendere i *software* e le applicazioni compatibili con i vari requisiti e bisogni degli

utenti, in conformità con le *Linee guida per l'accessibilità dei contenuti Web (WCAG)*. Questi *test* assicurano che le applicazioni siano adatte a tutti gli utenti e soddisfino le esigenze degli utenti con tutte le disabilità.
I *test di accessibilità* che sono stati fatti non erano richiesti dal proponente, infatti nascono da un'idea personale per conoscere il livello di accessibilità dell'applicazione. I *test* effettuati sono:

- verifica sul contrasto (bianco) dei colori presenti nell'applicazione, tramite l'utilizzo di *WebAIM*:
  - nero: 21:1. Il *test* è superato totalmente;
  - arancione: 2.57:1. Il risultato è basso, infatti non rispetta le *WCAG*;
  - verde scuro: 7.43:1. Il risultato è buono, infatti supera tutti i *test*;
  - rosso: 3.99:1. Supera i *test* per il formato di testo grande e per l'uso dei componenti grafici (es: *input form*), ma fallisce quello per il formato di testo medio.
- verifica che gli elementi fossero cliccabili e che le funzionalità implementate fossero utilizzabili attraverso uno *screen-reader*$_G$: entrambi superati utilizzando *NVDA*.

Il colore arancione, che non ha superato il *test*, non è stato modificato in quanto era richiesto dal proponente.

6.4 Test dei messaggi d'errore di un form

Per controllare il corretto funzionamento di tutti i componenti di un *form*, si è deciso di fare un *test* creando una nuova pagina contenente tutti i seguenti componenti: *GalDateEdit, GalNumericEdit, GalTextEdit e GalCheckBox*. Si sono creati i corrispettivi di ognuno per testare anche i valori *nullable*. In figura 6.1 viene mostrata la pagina di *test*: come si può vedere i messaggi d'errore cambiano se la varibile è *nullable* oppure no.

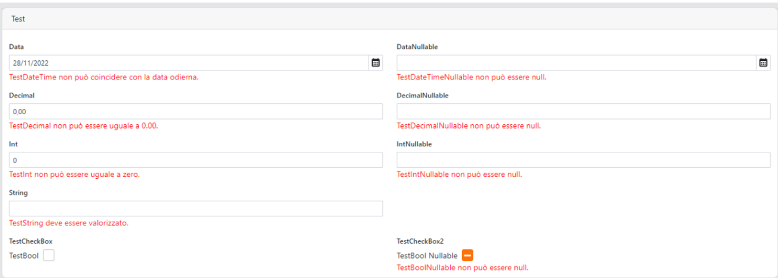


Figura 6.1: Visualizzazione dl test dei messaggi d'errore in un form

Il *form* è stato implementato nel seguente modo: ogni componente contiene un'identificativo (*Id*) per capire a quale *test* si riferisce e il *bind-Value* per prelevare dal *Dto* il giusto oggetto.
Nel metodo *OnInitializedAsync* viene creata una variabile locale *ErrorEmitter*<*T*>, dove *T* è la classe del *Dto*. Per ogni variabile del *Dto* viene eseguito un *if*: se le condizioni vengono rispettate viene invocato il metodo *Emit* con un'*espressione lambda*

e la stringa del messaggio d'errore. Infine al dizionario *errors* viene invocato il metodo *AddRange* per aggiungere il risultato di *errorEmitter*.
In figura 6.2 viene riportato un esempio dell'operazione di *testing* dei componenti.

```
if (testDto.TestInt == 0)
    errorEmitter.Emit(p => p.TestInt, "TestInt non può essere uguale a zero.");

if (testDto.TestIntNullable == null)
    errorEmitter.Emit(p => p.TestIntNullable, "TestIntNullable non può essere null.");

errors.AddRange(errorEmitter.GetResult());
```

Figura 6.2: Esempio di testing

Capitolo 7

Conclusioni

7.1 Raggiungimento degli obiettivi

Lo scopo del progetto di questo stage era realizzare e sviluppare, in parte, l'attuale applicazione *Match!* utilizzata dall'azienda. In particolare era richiesto l'implementazione di:

- Tre *widget* nella pagina principale "*Dati Generali*". Nel dettaglio era richiesto sviluppare un *donut chart,* un *point series chart* e un *widget* contenente l'elenco degli importi totali del cliente. Il contenuto di ogni *widget* doveva poter essere scelto dall'utente tramite un *popup*;

- Pagine contenenti una griglia, la quale doveva semplicemente includere dati tabellari, oppure includere anche operazioni di *new, change e delete* oppure contenere raggruppamenti in base ad un certo argomento;

- Pagine contenenti un *form*, con valori modificabili o meno.

Tutte le pagine contenenti griglie e *form* dovevano contenere negli opportuni campi identificativi (*Id*) lo stesso nome presente in *Match!*. In particolare le colonne delle griglie dovevano avere lo stesso tipo (es: *int, int?, bool, bool?, ecc*) delle colonne delle griglie di *Match!*.

7.2 Requisiti soddisfatti

In questo paragrafo viene dichiarato se i requisiti funzionali, qualitativi e di vincolo iniziali sono stati soddisfatti oppure no.

7.2.1 Requisiti funzionali soddisfatti

Nella tabella 7.1 vengono elencati tutti i requisiti funzionali della tabella 3.1 mostrando se sono stati soddisfatti oppure no.

Requisito	Descrizione	Stato
RFO1	L'utente per poter interagire con l'applicazione deve essere autenticato	Soddisfatto
RFO2	L'utente deve poter visualizzare il *widget* contenente il *donut chart* con uno degli importi	Soddisfatto
RFO3	L'utente deve essere in grado di esportare il *donut chart* in formato *.png*	Soddisfatto
RFO4	L'utente deve visualizzare un messaggio d'errore se durante il caricamento del *donut chart* c'è stato un problema	Soddisfatto
RFO5	L'utente deve poter visualizzare il *widget* contenente il *point series chart*	Soddisfatto
RFO6	L'utente deve essere in grado di esportare il *point series chart* in formato *.png*	Soddisfatto
RFO7	L'utente deve visualizzare un messaggio d'errore se durante il caricamento del *point series chart* c'è stato un problema	Soddisfatto
RFO8	L'utente deve poter visualizzare il *widget* contenente gli importi totali	Soddisfatto
RFO9	L'utente deve visualizzare un messaggio d'errore se durante il caricamento degli importi totali c'è stato un problema	Soddisfatto
RFO10	L'utente deve poter scegliere il contenuto di ognuno dei tre *widget* tra le scelte presenti	Soddisfatto
RFO11	L'utente deve visualizzare un messaggio se la pagina che ha cercato non esiste o è stata rimossa	Soddisfatto
RFO12	L'utente deve visualizzare un messaggio se non ha i permessi per navigare nella pagina cercata	Soddisfatto
RFO13	L'utente deve poter visualizzare e modificare alcuni valori dei *form*	Soddisfatto
RFO14	L'utente deve poter visualizzare la griglia ed inserire, modificare ed eliminare una riga da essa	Soddisfatto
RFD1	L'utente deve visualizzare il *loading* del *donut chart* se esso è in fase di caricamento	Soddisfatto
RFD2	L'utente deve visualizzare il *loading* del *point series chart* se esso è in fase di caricamento	Soddisfatto
RFD3	L'utente deve visualizzare il *loading* degli importi totali se essi sono in fase di caricamento	Soddisfatto
RFF1	L'utente deve visualizzare il *loading* all'interno della griglia, se essa è in fase di caricamento	Non soddisfatto

Tabella 7.1: Tabella del tracciamento dei requisti funzionali soddisfatti

Il requisito *RFF1*, come descritto in tabella, non è stato soddifatto, in quanto il componente *DxGrid* non dispone ancora di comandi per inserire all'interno della griglia ciò che si vuole; infatti si è notato che anche la scritta che compare quando non ci sono dati disponibili *"No data to display"* non può essere modificata. Essendo un componente recente (prima si utilizzava *DxDataGrid* ma *DevExpress* ha dichiarato

essere deprecata) non dispone ancora di alcune funzionalità.

7.2.2 Requisiti qualitativi soddisfatti

Nella tabella 7.2 vengono elencati tutti i requisiti qualitativi della tabella 3.2 mostrando se sono stati soddisfatti oppure no.

Requisito	Descrizione	Stato
RQO1	Le pagine implementate devono essere traducibili sia in italiano sia in inglese	Soddisfatto
RQD1	Deve essere effettuato un test per i *form* implementati	Soddisfatto

Tabella 7.2: Tabella del tracciamento dei requisiti qualitativi soddisfatti

7.2.3 Requisiti di vincolo soddisfatti

Nella tabella 7.3 vengono elencati tutti i requisiti di vincolo della tabella 3.3 mostrando se sono stati soddisfatti oppure no.

Requisito	Descrizione	Stato
RVO1	Per la realizzazione dell'applicazione deve essere utilizzato il *framework Blazor v. 3.2.0*	Soddisfatto
RVO2	Tramite l'utilizzo di *Blazor* deve essere utilizzato *ASP .NET 6*	Soddisfatto
RVO3	L'applicazione deve essere supportato nelle versioni recenti di *Google Chrome, Microsoft Edge e Mozilla Firefox*	Soddisfatto

Tabella 7.3: Tabella del tracciamento dei requisiti di vincolo soddisfatti

7.3 Valutazione personale

Lo *stage* ha avuto una durata (304 ore) coerente con il progetto aziendale previsto. Per la realizzazione di esso, ho eseguito inizialmente un ripasso della *programmazione ad oggetti* per poi proseguire con una dettagliata formazione sul *framework Blazor*. Grazie ad uno studio graduale e sempre più specifico sono riuscita ad apprendere i concetti in tempi ragionevoli. Grazie a questo progetto ho potuto conoscere e studiare *Blazor*, arricchendo le mie conoscenze della progettazione *web*. In particolare ho avuto modo di utilizzare il *framework ASP .NET* per l'implementazione delle pagine *web* e *DevExpress* per l'utilizzo di componenti già esistenti per creare una pagina, i quali hanno notevolmente aiutato a ridurre i tempi di sviluppo dell'applicazione.
In generale, ho avuto modo di imparare a gestire da sola tutte le fasi dell'applicazione: infatti si è iniziato dall'*analisi dei requisiti* fatta con il *tutor* aziendale, proseguendo poi con la progettazione e lo sviluppo, terminando infine con una parte ristretta di *test*.

Glossario

Api: acronimo di *Application Programming Interface API* ("interfaccia di programmazione di un'applicazione") indica ogni insieme di procedure disponibili al programmatore, di solito raggruppate a formare un set di strumenti specifici per l'espletamento di un determinato compito all'interno di un certo programma La finalità è ottenere un'astrazione, di solito tra l'hardware e il programmatore o tra software a basso e quello ad alto livello semplificando così il lavoro di programmazione

Blazor: framework web gratuito e open source, creato da Microsoft, che permette di creare un'applicazione web usando il linguaggio *C#* ed *HTML*

Ciclo di vita: sono i metodi che gestiscono il ciclo di vita di vita di un componente *Blazor*. I più noti sono *OnInitialized, OnInitializedAsync, OnParametersSet, SetParametersAsync, ecc.*

Deserializzazione: è un processo tramite il quale un oggetto viene ricostruito a partire dalla sua rappresentazione binaria ottenuta da un *file* o da un canale di rete.

DOM: acronimo di *Document Object Model*, definisce la struttura logica del documento e il modo in cui si può accedere e modificare il documento stesso. Il principale linguaggio di programmazione usato nel *DOM* è *JavaScript*

Garbage Collection: letteralmente "raccolta di rifiuti", si intende una modalità automatica di gestione della memoria, mediante la quale un sistema operativo, o un compilatore e un modulo di *run-time* liberano porzioni di memoria non più utilizzate dalle applicazioni.

Outsourcing: letteralmente "esternalizzazione", è un servizio che prevede di affidare ad una azienda esterna, tramite uno specifico contratto, tutti i servizi IT: dalle risorse tecniche a quelle umane

Rendering: letteralmente "restituzione grafica", identifica il processo di generazione di un'immagine a partire da una descrizione matematica di una scena tridimensionale, interpretata da algoritmi che definiscono il colore di ogni punto dell'immagine digitale.

Screen-reader: è una forma di tecnologia assistiva che identifica ed interpreta il testo mostrato sullo schermo di un *computer*, presentandolo come *output* in sintesi vocale.

Serializzazione: è un processo per salvare un oggetto in un supporto di memorizzazione lineare, o per trasmetterlo su una connessione di rete. La serializzazione può essere in forma binaria o può utilizzare codifiche testuali direttamente leggibili.

Singleton: è un *design pattern creazionale* che ha lo scopo di garantire che di una determinata classe venga creata una e una sola istanza, e di fornire un punto di accesso globale a tale istanza.

SGQ: raccolta di politiche, processi, procedure documentate e registrazioni che definiscono le regole interne che delimitano il modo in cui l'azienda crea e fornisce il prodotto o il servizio ai clienti

UI: Acronimo di *User Interface* ("interfaccia utente"), è l'interfaccia uomo-macchina, cioè ciò che permette all'uomo di interagire con la macchina e viceversa

UML: acronimo di *Unified Modeling Language*, è stato creato per realizzare un linguaggio di modellazione visivo comune, ricco sia nella semantica che nella sintassi, per l'architettura, la progettazione e l'implementazione di sistemi *software* complessi sia dal punto di vista strutturale che comportamentale.

Use Case: letteralmente "Casi d'uso", sono una tecnica usata in informatica nei processi di Ingegneria del Software per eseguire in maniera esaustiva la raccolta dei requisiti

VPN: Acronimo di Virtual Private Network, crea una connessione sicura tra un dispositivo (ad esempio uno *smartphone o laptop*) e *Internet.* La *VPN* ti permette di inviare i dati attraverso un *tunnel* sicuro e criptato a un *server* esterno: il *server VPN.* Da lì i dati sono inviati alla loro destinazione su *internet.*