Università Politecnica delle Marche

FACOLTÀ DI INGEGNERIA
Corso di Laurea in Ingegneria Informatica e dell'Automazione

Componenti di un'interfaccia per un'applicazione di gestione del personale basati sul framework Angular

Components of an interface for a staff management application based on the Angular framework

Candidato:
Lorenzo Fratini

Relatore:
Prof. Alessandro Cucchiarelli

Anno Accademico 2019–2020

Indice

Abstract

La seguente tesi descrive il lavoro svolto durante il periodo di tirocinio curriculare presso Pluservice S.r.l., un'azienda leader nel mercato italiano per lo sviluppo di sistemi informativi integrati (ERP) per le aziende di trasporto passeggeri. Fra gli applicativi interni all'azienda vi è Motus, un prodotto utilizzato per gestire l'intero processo organizzativo di un'azienda di trasporto pubblico passeggeri, sia in ambito urbano che extraurbano. Data la numerosità delle funzionalità interne a questa applicazione, la seguente tesi descrive solo alcune di esse. In particolare, si fa riferimento allo sviluppo di alcune componenti per la gestione dei dipendenti attraverso Angular, un noto framework JavaScript utilizzato per sviluppare applicazioni Web. La realizzazione di queste funzionalità avviene definendo una serie di sottomaschere, ognuna delle quali è caratterizzata da meccanismi e automatismi, appartenenti alla maschera del dipendente. In particolar modo si descrive la sottomaschera che contiene le funzionalità che permettono di gestire i dati aziendali e anagrafici, i quali sono modificabili dall'operatore che utilizza la maschera stessa. Poi si procede andando ad illustrare la sottomaschera contenente i dati aziendali storicizzati in una serie di tabelle ognuna delle quali è associata ad un particolare dato. Anche in questo caso l'operatore può modificare le righe già esistenti delle tabelle oppure provvedere ad aggiungerne delle nuove. Quindi, data la possibilità di effettuare delle operazioni che possono alterare i dati già esistenti nella maschera del dipendente, è necessario sviluppare un meccanismo di salvataggio che permetta di conservare le modifiche effettuate. Infine si descrive la maschera contenente i KPI (*Key Performance Indicators*), che rappresentano gli indici di un processo aziendale e che vengono mostrati mediante una tabella dove ad ogni voce sono associate delle statistiche che possono essere visualizzate graficamente in una sezione apposita della medesima maschera.

Capitolo 1

Introduzione

Negli ultimi anni si è sviluppato nel contesto internazionale un crescente interesse nell'esigenza di coordinare in una struttura unitaria le informazioni relative al sistema dei trasporti, provenienti da svariate fonti.
Tale interesse deriva direttamente dalla necessità di gestire e programmare in modo efficiente il sistema dei trasporti da parte degli organismi preposti, siano esse amministrazioni pubbliche o aziende private.
In questo contesto trova spazio l'*Intelligent Trasport Systems* (ITS), un termine che sostanzialmente rappresenta lo sforzo di introdurre l'ICT (*Information and Communication Technology*) nell'ingegneria del trasporto.
L'ITS nasce inizialmente dall'esigenza di gestire i problemi causati dalla congestione del traffico, che riduce l'uso delle infrastrutture dei trasporti ed aumenta i tempi di percorrenza, il consumo di carburante e l'inquinamento. Inoltre, nel tempo si è aggiunta quella della sicurezza, che ha dato origine alla sorveglianza stradale, ai limiti di velocità monitorati, ai semafori intelligenti e a molte altre soluzioni.
Dopo queste prime evoluzioni delle infrastrutture si può notare come l'ITS sia entrato anche all'interno dei veicoli, grazie alla nascita di sistemi di navigazione satellitare GPS, alle comunicazioni wireless veicolo-veicolo e veicolo-ambiente e ai controlli in tempo reale.
Tra i campi in cui le soluzioni ITS sono adottate, vi sono gli *Advanced Public Transport System* (APTS), cioè servizi per la gestione del trasporto pubblico dei passeggeri.
In questo contesto trova spazio **Pluservice S.r.l**, un'azienda italiana responsabile

della produzione di sistemi informativi integrati (ERP) per le aziende di trasporto passeggeri. Tra i suoi applicativi può vantare **Motus**, che rappresenta un prodotto utilizzato per gestire l'intero processo organizzativo di un'azienda di trasporto pubblico passeggeri, sia in ambito urbano che extraurbano.
Tra le diverse funzionalità che Motus presenta si ha la gestione della rete, delle rotazioni tra gli autisti, l'inserimento delle tabelle orarie, la progettazione dei turni veicolo/uomo e l'utilizzo di un algoritmo di ottimizzazione per ottenere turni migliori.
Il seguente progetto è incentrato sullo sviluppo di una porzione di questa applicazione, in particolar modo si porrà l'attenzione sulla figura del dipendente e su parte delle problematiche che l'applicazione dovrà gestire a questo riguardo. Più precisamente si farà riferimento alla maschera associata al dipendente la quale sarà caratterizzata da una serie di voci con cui l'utente può attivare una serie di funzionalità ed automatismi. Infatti, non si presenterà come una semplice maschera relativa all'inserimento dei dati, ma si dovranno gestire le possibili interazioni che l'operatore che usa l'applicazione potrà avere, come ad esempio la modifica dei dati anagrafici del dipendente oppure la gestione dei relativi dati aziendali. In particolar modo, visto che quest'ultimi saranno storicizzati opportunamente in delle tabelle in una sezione apposita, si dovrà anche gestire l'eventuale possibilità di modificare questi dati già esistenti oppure aggiungerne dei nuovi. Inoltre, a causa di queste operazioni si presenta anche il problema di poter conservare il risultato di queste interazioni, per cui si dovrà implementare un meccanismo di salvataggio dei dati. Oltre questi aspetti non si possono escludere la gestione dei *Key Perfomance Indicators*) (KPI) che rappresentano un insieme di indicatori che fungono da indici dell'andamento di un processo aziendale. Data la loro importanza sarà necessario monitorarli relativamente ad ogni dipendente dell'azienda. Inoltre, per facilitarne la loro visualizzazione e analisi si mostrerà il loro andamento graficamente a seconda dell'indicatore selezionato.

Le tecnologie che verranno utilizzate per implementare tutto ciò saranno quelle più moderne, non a caso verrà utilizzato principalmente *Angular*, un framework

open-source largamente adoperato per lo sviluppo di *single-page applications.*[1] A supporto di questa tecnologia ne sono state utilizzate altre tra cui *PrimeNG*, una libreria open-source di componenti UI di Angular, e *Lodash*, una libreria JavaScript che fornisce funzionalità che facilitano la gestione di strutture dati complesse.
Quindi, come si evince da quanto detto sopra, lo scopo del lavoro illustrato in questa tesi è implementare una serie di funzionalità e automatismi per la gestione del personale all'interno di un'azienda di trasporto pubblico di passeggeri utilizzando gli strumenti forniti dal framework Angular.

[1]Una ***Single-page application*** è un'applicazione in cui tutto il codice necessario (HTML, CSS e JS) è recuperato in un singolo caricamento della pagina e le risorse sono caricate dinamicamente e aggiunte a quella corrente quando necessario. Questo è possibile perchè lo sviluppo dell'interfaccia utente non è gestito lato server, bensì lato client.

Capitolo 2

Contesto Applicativo

2.1 Sistemi informativi integrati ERP

I sistemi informativi integrati **ERP** (**Enterprise Resource Planning**) rappresentano soluzioni applicative concepite in modo da integrare su base aziendale i processi operativi ed amministrativi che regolano lo svolgersi delle varie attività aziendali, riconducendo in un unico schema logico i flussi di informazione che accompagnano le transazioni operative originatesi in diverse aree funzionali.

La traduzione letterale di ERP sarebbe: "sistemi per la pianificazione aziendale delle risorse", ma non è quella corretta.
La traduzione nella nostra lingua più consona, anche se non letterale, sarebbe: "sistemi informativi integrati", ad indicare proprio la capacità di questi sistemi di integrare i diversi sottoinsiemi informativi presenti in azienda. La mancata duplicazione dei dati presenti nel sistema e la diffusione istantanea delle informazioni consente liberare risorse precedentemente impegnate in attività a nessun valore aggiunto e, spesso, con scarso contenuto professionale, tipiche dei sistemi non integrati. L'eliminazione di duplicazioni nelle attività svolte non solo consente risparmi di costo, ma determina anche riduzione nei tempi di attesa e dunque, accorciamento dei tempi di ciclo, una maggiore flessibilità e prontezza di risposta.

I sistemi informativi integrati, basandosi su un unico database condiviso, permettono, ad ogni utente autorizzato di accedere alle informazioni, a prescindere dalla

sua collocazione organizzativa o localizzazione fisica.

Un'altra caratteristica fondamentale dei sistemi integrati ERP è la modularità che riguarda la loro architettura, organizzata rispetto ad una gerarchia di sistemi applicativi ciascuno posto a presidio di aree funzionali ed attività operative specifiche. Ad ogni area aziendale corrisponde, in genere, un vero e proprio modulo o applicazione. I diversi moduli sono progettati in un'ottica gestionale integrata del lavoro e sono quindi in grado di supportare gli utenti del sistema in tutte le attività richieste da processi aziendali complessi. Questa caratteristica permette di scegliere un percorso di copertura incrementale delle varie aree aziendali con una gestione sicuramente più efficiente da un punto di vista interno dell'azienda.[1]

2.2 Sistemi di trasporto intelligenti

L'integrazione delle Tecnologie dell'Informazione e della Comunicazione (*Information & Communication Tecnology, ITC*), con l'ingegneria dei trasporti, ha dato origine ai cosiddetti **Sistemi di Trasporto Intelligenti** (***Intelligent Transportation System, ITS***).
Gli ITS sono quel complesso di tecnologie, procedure e strumenti che consentono di migliorare il trasporto delle merci e la mobilità delle persone, grazie al controllo e alla gestione efficiente degli utenti del sistema di trasporto.
Il funzionamento degli ITS si basa su un monitoraggio continuo del sistema di trasporto per raccogliere, integrare ed elaborare i dati relativi a veicoli, infrastrutture e altre risorse, al fine di generare e diffondere informazioni utili all'utenza e agli operatori.
Tra i vari campi in cui le soluzioni ITS sono comunemente adottate, vi è quello per la gestione dei servizi di trasporto pubblico dei passeggeri (*Advanced Public Transport System, APTS*).
Questi sistemi trovano applicazioni in ogni diversa fase del ciclo di vita di un servizio di trasporto, ad esempio consentono di:

- **Offrire un servizio di trasporto pubblico di qualità**

  Assicurando maggiore regolarità, puntualità, affidabilità e prontezza del ser-

vizio alle esigenze di mobilità degli utenti con la prevenzione e la gestione di eventuali disservizi;

- **Migliorare la sicurezza**

  Fornendo assistenza al personale e ai passeggeri, ad esempio attuando misure di pronto intervento in caso di emergenze ed incidenti;

- **Potenziare l'offerta**

  Fornendo servizi aggiuntivi al cittadino, come un'informazione in tempo reale circa lo stato di servizio delle linee e i tempi di arrivo dei mezzi mediante applicazioni per smartphone e display installati alle fermate;

- **Certificare la consuntivazione**

  Acquisendo il chilometraggio effettuato e il rispetto del programma di esercizio, raccogliendo una grande quantità di informazioni per ciascun mezzo come: cose perse, anticipo/ritardo sulla tabella di marcia, etc.

- **Diminuire i costi di produzione del servizio**

  Ottenendo risparmi attraverso l'ottimizzazione delle risorse umane e dei mezzi di trasporto, grazie ai dati acquisiti che consentono di rivedere quanto stabilito in fase di pianificazione e programmazione. [2]

L'architettura base di un sistema di gestione del trasporto pubblico è mostrata in *Figura 1.*

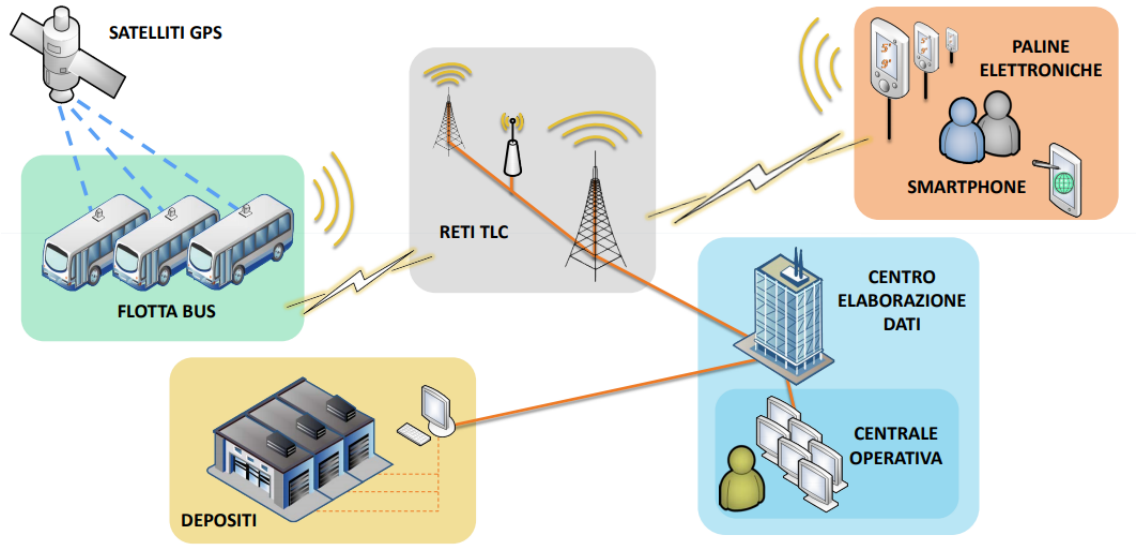

Figura 1: Architettura APTS

In questa architettura la Centrale Operativa supervisiona e interviene direttamente sulla gestione del servizio occupandosi del monitoraggio delle linee, della gestione delle comunicazione con la flotta, degli eventi critici, dell'informazione dinamica alla clientela (mediante paline elettroniche, monitor di bordo e applicazioni per smartphone) e della gestione dei dati di consuntivo.
Nel sistema di bordo è registrato il servizio di trasporto da effettuare. Quando vi è uno scostamento fra il servizio programmato e quello effettuato il sistema lo rileva, avvisa il conducente ed invia l'informazione all'operatore della Centrale Operativa.

2.3 Motus

2.3.1 Introduzione

Tra le aziende che si occupano di fornire questa tipologia di servizi vi è Pluservice, un'azienda del territorio di Senigallia, leader nel fornire sistemi informativi integrati (ERP) per le aziende di trasporto passeggeri.
Tra i suoi prodotti può vantare Motus, un applicativo per gestire l'intero processo organizzativo di un'azienda di trasporto pubblico di passeggeri, sia in ambito urbano che extraurbano e relativamente alle varie modalità di trasporto stradale, ferroviario e navigazione.

L'applicativo comprende diverse funzioni, tra cui:

- Gestione della rete, con la possibilità di utilizzare degli strumenti per effettuare delle simulazioni;
- Inserimento delle tabella orarie, con report per autisti, pubblico e operatori;
- Progettazione dei turni veicolo/uomo nel rispetto delle normative e dei contratti aziendali;
- Utilizzo dell'algoritmo di ottimizzazione dei servizi per ottenere turni migliori per l'autista e in termini economici per l'azienda;
- Gestione delle rotazioni, per organizzare in maniera semplice ed intuitiva la turnazione, le variazioni giornaliere rispetto a quanto programmato e la relativa consuntivazione alle paghe o al controllo di gestione.

La base di dati a cui l'applicazione fa riferimento è stata progettata secondo lo standard di riferimento europeo Transmodel [1] e garantisce soluzioni adeguate alle esigenze di ogni operatore del settore. I vantaggi che offre la soluzione Motus sono:

- **Ambiente multi aziendale**, cioè il sistema gestisce in modo sicuro gli ambienti multi aziendali garantendo la visibilità dei dati in maniera gerarchica e profilata secondo le esigenze dei vari operatori e delle autorità di controllo;
- **Configurazione utenti**, cioè ogni utente è configurato in modo specifico per operare all'interno dell'applicativo;
- **Soluzioni web e modulari**, grazie ad una architettura e una veste grafica web arricchita da strumenti innovativi quali: tabelloni capaci di gestire i dati in maniera intuitiva e veloce, grafici che rappresentano visivamente lo stato dei dati e ne consentono un rapido controllo

[1] Transmodel (formalmente *CEN TC278, Reference Data Model For Public Transport, EN 12896*) è lo standard europeo nel trasporto pubblico di persone che offre un modello di dati di riferimento per un sistema informativo integrato del trasporto pubblico, utilizzabile liberamente dalle imprese operanti nel settore.

- **UX/UI**, cioè le soluzioni proposte sono tutte realizzate secondo i moderni principi di User Experience e User Interface (UX/UI) per rendere semplice ed immediato l'utilizzo da parte degli operatori;
- **Cartografia**, grazie al modulo cartografico vettoriale integrato all'interno di Motus
- **Parametrizzazione contrattuale di primo e secondo livello**, per calcolare in modo automatico i dati da inviare al software che gestisce le paghe riducendo al minimo l'intervento degli operatori;
- **Controllo in tempo reale sul rispetto delle normative** per quanto riguarda i tempi di guida, gli straordinari, i riposi giornalieri e settimanali;
- **Controllo totale di tutte le diverse fasi** sia in termini economici che in termini di efficienza del servizio svolto.

L'architettura dell'applicazione è schematizzata in *Figura 2*:

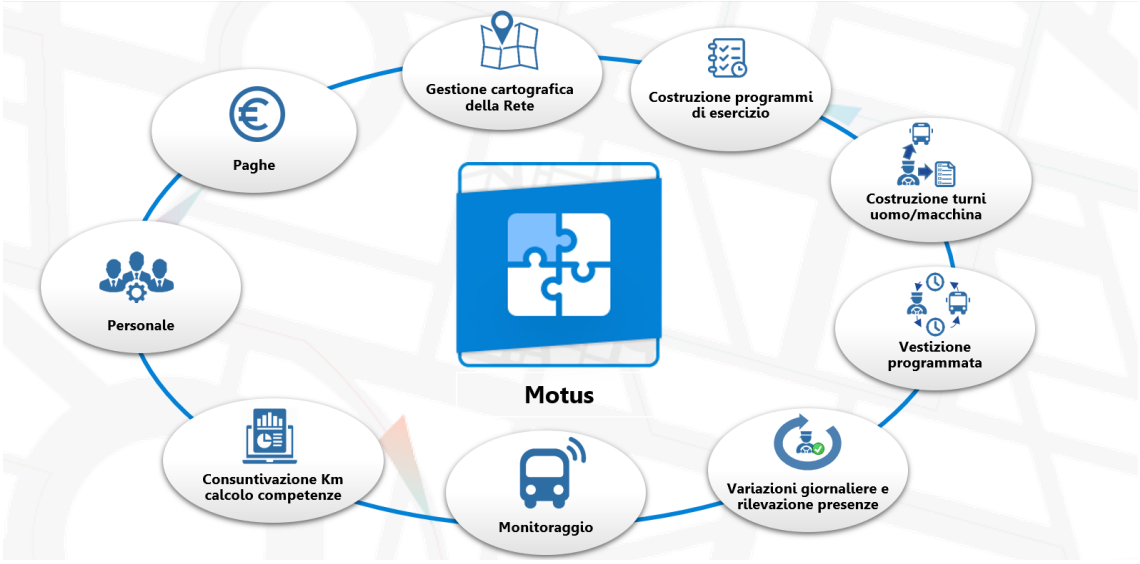

Figura 2: Architettura Motus

L'applicativo è parametrizzato al fine di mostrare in apertura a ciascun operatore soltanto la home con i moduli e le singole maschere a lui abilitati, tra le quali l'operatore si può spostare mediante un menù ad albero.
Tra le varie anagrafiche necessarie al funzionamento dell'applicativo spicca sicuramente la form del Dipendente che non si può considerare come una semplice maschera di inserimento.
Oltre ai dati amministrativi di ogni dipendente, quali la residenza, la data di nascita, etc., comprende anche tutti i dati che riguardano la sua storia lavorativa e ne fornisce una consultazione rapida e puntuale grazie alla loro storicizzazione. Infatti, nelle aziende di trasporto è necessaria molta flessibilità, a cause del fatto che un'autista cambia spesso mansione, qualifica o area di lavoro oppure possiede più patenti/abilitazioni che lo rendono disponibile o indisponibile allo svolgimento di alcuni turni piuttosto che altri.
La form contiene anche una sezione di KPI[2] con griglie e grafici che facilita l'operatore nella lettura di alcuni dati riepilogativi (parametrizzabili) che poi gli saranno necessari per fare delle variazioni giornaliere di servizio in modo efficiente e il meno discrezionale possibile evitando quindi il malcontento e le lamentele da parte degli autisti.

2.3.2 Motus - Modulo Cartografico

L'accesso al modulo cartografico avviene scegliendo la versione di rete di lavoro.
La gestione delle versioni consente di creare più reti per gestire in anticipo eventuali variazioni alla viabilità o simulare la costruzione di nuove versioni dei percorsi e verificarne l'impatto nell'esercizio senza intaccare l'ambiente di produzione e senza ricaricare manualmente tutti dati delle tabelle orarie e dei servizi veicolo/uomo, in quanto il sistema esegue le sostituzioni automaticamente.
É possibile utilizzare come stradario Tom Tom, Open Street Map (OSM) o qualunque altro stradario fornito dal cliente implementato secondo il modello su grafo.
Direttamente da questo modulo è possibile, mediante un menù:

[2]KPI (*Key Performance Indicator*) sono indicatori che riflettono i fattori critici di successo per un'organizzazione, usati per misurare i risultati conseguiti dall'organizzazione medesima

- **Gestire le fermate**, cioè i punti in cui è prevista la fermata del mezzo per la salita o la discesa dei passeggeri. Ciascuna fermata, che tipicamente è collocata ai lati della strada, viene proiettata nell'asse stradale grazie ad un apposito punto di proiezione di fermata.

- **Gestire delle tratte**, cioè la gestione degli archi fermata che rappresenta la congiunzione monodirezionale di due fermate ottenuta mediante successione ordinata di archi stradali. Per questo un arco fermata è composto da due elementi principali: la fermata di partenza e di arrivo e il tragitto da percorrere per effettuare la congiunzione tra le suddette fermate.

- **Gestione dei percorsi**, il che consente l'implementazione della rete di trasporto pubblico locale. La definizione di un percorso è composta dal dettaglio delle fermate servite e dalla relativa viabilità percorsa per mezzo dell'elenco degli archi fermata utilizzati.

2.3.3 Motus - Time Table

Completata la parte cartografica, si passa all'inserimento delle corse nelle tabelle orarie.
Un link orario rappresenta una tratta o un insieme di tratte a cui va associato un tempo. Un valore generico di default è obbligatorio ed è già popolato in base ai parametri cartografici. Possono comunque essere definite più tempistiche, dato che la percorrenza di un link orario è variabile non solo rispetto ai km/velocità commerciale, ma anche alla fascia oraria della giornata (notturna, bassa, ecc..), a particolari condizioni traffico/servizio (esempio festivo, giorno di mercato, ecc..) o alla tipologia del mezzo utilizzato.
La struttura prevede l'inserimento del tempo di sosta, che è un valore fisso che fa slittare i tempi delle fermate successive, o di un tempo di transito, che invece è dinamico e indipendente dall'orario di arrivo alla fermata e ne fissa l'obbligo a non ripartire prima di quel tempo.
Una corsa rappresenta l'effettuazione di un particolare percorso ad un orario di partenza a cui va associata una particolare validità. Le corse sono raggruppate in tabelle orarie che sono legate, oltre che alla linea, alla rete cartografica che ne

identifica quindi la composizione del percorso. L'inserimento dell'ora di partenza popola gli orari di passaggio ad ogni fermata che sono calcolati sulla base delle temporizzazioni inserite ed è il sistema a capire se la corsa inizia nella temporizzazione giornaliera e passa poi alla notturna.

2.3.4 Motus - Creazione Turni

In questa sezione si associano le corse ai diversi veicolo e autisti.
Riguardo la scelta del veicolo, è previsto l'interfacciamento con l'algoritmo di ottimizzazione dei servizi che in modo automatico è in grado di generare velocemente soluzioni di qualità migliore di quelle ottenute dagli operatori più esperti, di poter simulare scenari normativi alternativi in sede di contrattazione sindacale, di analizzare realtà diverse al fine di un contenimento del costo del lavoro e di fornire un adeguamento ai veloci cambiamenti di scenario del mercato.
Il successivo passaggio consiste nella scelta dell'autista ed è prevista una sezione apposita in cui si associano le corse all'autista partendo dalle tabelle orarie e dai turni dei veicoli precedentemente creati così da poterli assegnare in maniera più veloce a blocchi precaricati, interfacciandosi sempre con la procedura di ottimizzazione dei turni.
Completata la costruzione dei servizi veicolo e uomo, vengono creati i Set di Produzione.
Un Set è un raggruppamento di turni veicolo e uomo, omogeneo per il tipo giorno che va pianificato in un calendario. L'associazione del set al calendario è un processo automatico, cioè il sistema in base ai giorni operativi calcolati e ai giorni liberi nel calendario propone la pianificazione che l'operatore può variare o confermare.
Il set può essere di due tipologie: di produzione o di simulazione. Se il set è di tipologia produzione, la sua pianificazione comporta dei blocchi in tutto quello che viene definito a monte, ovvero tabelle orarie e servizi che non saranno più modificabili in quelle date. Nel caso invece di set di tipologia simulazione la pianificazione non comporta alcun blocco delle attività precedenti.

2.3.5 Motus - Rostering e assegnazione

Per l'associazione del servizio programmato (turni di lavoro e riposi) ai dipendenti o ai veicoli vengono innanzitutto definite delle matrici di rotazione che sono raggruppamenti di dipendenti (o veicoli) che svolgono nel tempo gli stessi turni seguendo una sequenza periodica di turnazione. Una matrice può essere:

- Libera, ossia l'operatore scrive l'intera sequenza e poi la replica con uno scostamento per tutte le righe successive della matrice.
- Con ciclo di riposi: può essere usato un ciclo di riposi che è condiviso con altre matrici. Un esempio classico è la matrice dei riposi a scalare, dove i turni dei dipendenti variano fra di loro, ma viene condivisa come logica di fondo la ripetizione del turno e soprattutto il rispetto della sequenza dei riposi, fra un periodo e l'altro di turnazione.
- Matrici fisse (ad esempio per il personale ufficio o per i veicoli).

Qualora ci dovessero essere delle modifiche rispetto a quanto stabilito si può utilizzare un servizio apposito per la pianificazione, rappresentato da un tabellone con una visualizzazione periodica o programmata sulla singola giornata, che permette di gestire quotidianamente i cambi turni uomo-veicolo, aggiungere dei servizi estemporanei, gestire le assenze e controllare gli straordinari.
L'intero sistema è collegato in tempo reale con altri applicativi Pluservice attraverso i quali si colloquia per quanto riguarda le richieste di ferie e di cambi turno degli autisti e per ricevere informazioni sui fermi macchina o anomalie sui mezzi e le conferme dei piani di viaggio.

Capitolo 3

Obiettivi del progetto

3.1 Introduzione

Motus rappresenta un'applicazione che permette di gestire interamente tutte le problematiche che un'azienda di trasporto pubblico deve affrontare quindi, data la complessità di tale applicazione, nello sviluppo del progetto a cui questa relazione fa riferimento si è trattato solo una porzione di esso.
Ciò non vuol dire che si sono solo sviluppate semplici maschere relative all'inserimento di dati, perché il fine del lavoro è stato quello di creare degli automatismi dovuti alle interazioni di tali dati.
In particolar modo, la problematica che sarà presa in esame sarà stata lo sviluppo di alcune componenti riguardanti la maschera del Dipendente, ovvero quella che poi utilizzerà l'operatore per accedere ai dati personali e aziendali di un certo dipendente, nonché per poter visualizzare alcuni di essi in via grafica così da facilitarne la relativa analisi. La maschera del Dipendente presenterà al suo interno diverse componenti. In particolare, fanno parte di essa:

- la sezione dei dati anagrafici e dei dati aziendali;
- la sezione dei dati di guida, che conterrà tutti i dati delle patenti dell'autista con le relative scadenze;
- la sezione degli storici, contenente tutti i dati relativi alla situazione aziendale storicizzati;

- la sezione delle abilitazioni, in cui saranno presenti tutte le abilitazioni del dipendente;
- la sezioni "Voci ad personam", che rappresenta una serie di voci che possono essere assegnate di default al dipendente;
- la sezione delle scadenze, che rappresenta una griglia con una serie di scadenze e la relativa data;
- la sezione delle ferie e dei permessi che permetterà un riepilogo del loro stato;
- Una sezione relativa ai documenti che rappresenta una griglia che contiene qualsiasi documenti caricato dall'operatore;
- la sezione KPI con tutti gli indicatori che riguarderanno il dipendente

Di tutte queste sezioni, quelle che saranno prese come riferimento per lo sviluppo del progetto saranno la sezione contenente i dati anagrafici e aziendali, quella che fa riferimento agli storici e quella contente i KPI e, ovviamente, il menu di navigazione che permetterà poi di navigare fra di esse.

3.2 Dati anagrafici e aziendali

Riguardo la sezione contenente i dati anagrafici e aziendali essa mostrerà in una prima parte i dati personali del dipendente, quindi il suo nome, il suo cognome, la sua età, il suo sesso, etc., con la possibilità di inserire anche una foto, e in una parte sottostante una serie di campi, che faranno riferimento ai dati aziendali, come ad esempio la mansione del dipendente, che dovranno essere popolati con il valore più recente del rispettivo storico.
Questi campi non dovranno essere inseriti manualmente, ma l'obiettivo che si è imposto in fase di progettazione è stato che al caricamento della maschera del dipendente la prima sezione che si dovrà aprire sarà quella contenente i dati anagrafici e aziendali. Tutte le caratteristiche proprie del dipendente faranno parte, lato back-end, di tabelle specifiche e quindi, ogni volta che si dovrà far rifermento a dati contenuti in queste strutture dati, si dovrà sviluppare un servizio Web apposito che tramite

una chiamata GET andrà ad estrarre i valori di interesse. Altra caratteristica fondamentale è rappresentata dal fatto che l'operatore che utilizzerà la maschera potrà liberamente modificare ogni campo contenente questi dati personali del dipendente, per cui, affinché tutte le modifiche possano essere conservate in modo permanente anche dopo un nuovo caricamento della pagina web, sarà necessario implementare un meccanismo di salvataggio che poi andrà ad aggiornare i dati contenuti nelle rispettive tabelle del database mediante una chiamata di tipo POST.
La parte sottostante contenente i dati aziendali, come già specificato precedentemente, conterrà una serie di campi che poi dovranno essere riempiti con i valori più recenti contenuti nelle rispettive tabelle della sezione degli storici, i cui requisiti funzionali verranno analizzati successivamente.
Inoltre, questa sezione non potrà essere modificata dall'operatore che utilizzerà la maschera, come quella con i dati personali del dipendente, ma sarà di sola lettura e l'unico modo per poter cambiare i contenuti dei campi sarà agire direttamente sulle tabelle da cui i dati provengono.
In *Figura 3* vengono mostrati schematicamente i requisiti funzionali descritti precedentemente.

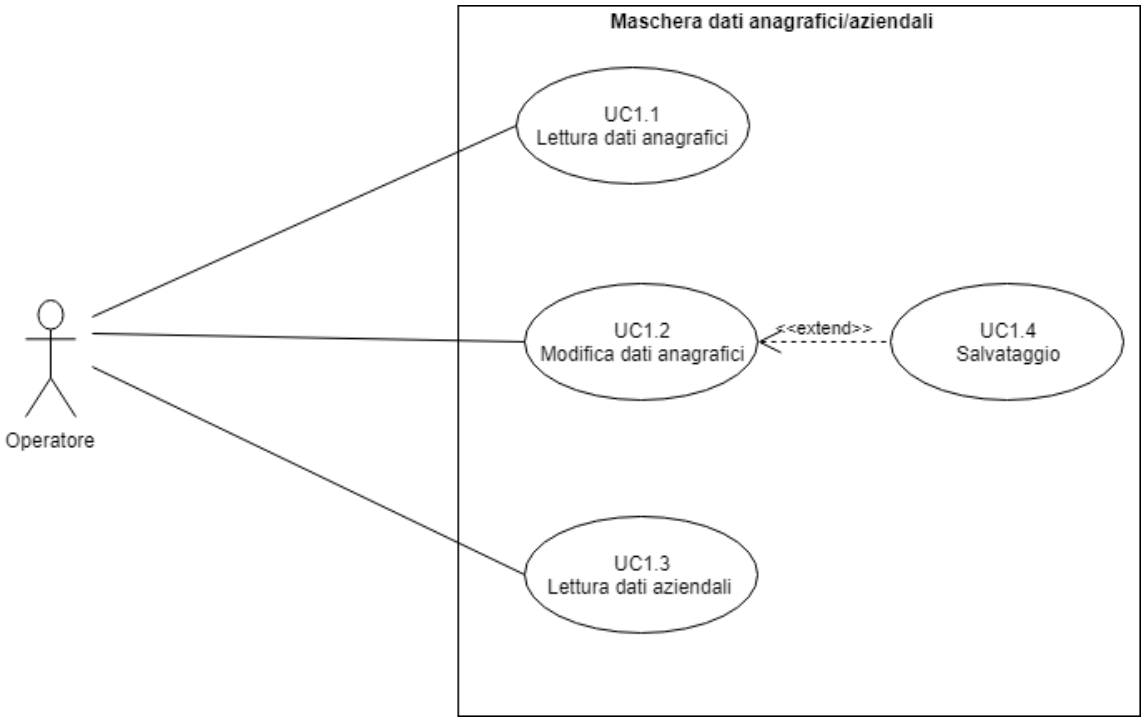

Figura 3: Diagramma dei casi d'uso della maschera contenente i dati anagrafici e aziendali

3.3 Storici

La successiva maschera a cui si farà riferimento sarà quella contenente i dati relativi agli storici del dipendente.
Rappresenta una sezione che conterrà diverse tabelle ognuna delle quali farà riferimento a particolari classi di dati aziendali, così da facilitarne poi l'analisi e la visualizzazione.
Innanzitutto, il primo requisito di questa maschera sarà realizzare un menu a scomparsa in modo che, al caricamento iniziale della maschera del dipendente, cioè quando vengono visualizzati i suoi dati anagrafici e aziendali, oppure nel momento in cui si selezionerà una sezione che non sarà quella a cui si fa riferimento in questo caso, nel menu laterale di navigazione compaia solo la voce "Storici", mentre nel momento in cui si cliccherà su di essa, comparirà un menu a scomparsa nel quale sarà possibile selezionare o deselezionare delle voci che determineranno la comparsa o la scomparsa delle rispettive tabelle dalla griglia, facilitando l'operatore nella visualizzazione e analisi dei dati contenuti in essi senza andare a ricercare la tabella d'interesse con la barra di scorrimento verticale.
Le tabelle che faranno parte di questa sezione conterranno dati circa le unità organizzative, i contratti, l'area di lavoro, le mansioni, le qualifiche, le residenze, le matricole, i badge, i rapporti, i centri di costo e i livelli e per ognuna di esse i record avranno in comune campi che si riferiranno ad un codice che permetterà di identificare la riga e un riferimento ad una data iniziale e finale.
Anche in questa situazione, in modo simile a quanto fatto nella sezione contenente i dati anagrafici e aziendali, sarà necessario andare a popolare i campi di queste tabelle attraverso delle strutture dati che si otterranno mediante opportune funzioni di filtraggio con le quali si andranno ad estrarre, dalla totalità dei dati riferiti al dipendente, solo quelli relativi alla tabella che si vorrà popolare. Tuttavia, in questa circostanza oltre a dover gestire la possibilità che l'operatore possa modificare i record delle tabelle, è stato imposto come ulteriore requisito funzionale anche la gestione di un eventuale inserimento di un record in ogni tabella.
Nel primo caso per ogni riga sarà possibile modificare la sola data di inizio e di fine, attraverso un doppio click nel record di interesse. Questo evento porterà all'apertura di un wizard tramite il quale sarà possibile realizzare le operazioni di modifica

desiderate.

Invece, nel caso in cui l'operatore voglia inserire un nuovo record verrà mostrato sempre un wizard nel quale però oltre alle date di inizio e di fine dovrà anche popolare i restanti campi da un set predefinito.

In entrambe le situazioni non si dovrà dare la possibilità di conferma della modifica o dell'inserimento, mediante la visualizzazione di messaggi di errore attraverso un pop-up, qualora l'operatore non inserisca alcun valore in almeno uno dei campi delle date oppure inserisca un valore della data di inizio successivo a quello della fine.

In *Figura 4* vengono mostrati schematicamente i requisiti funzionali descritti precedentemente.

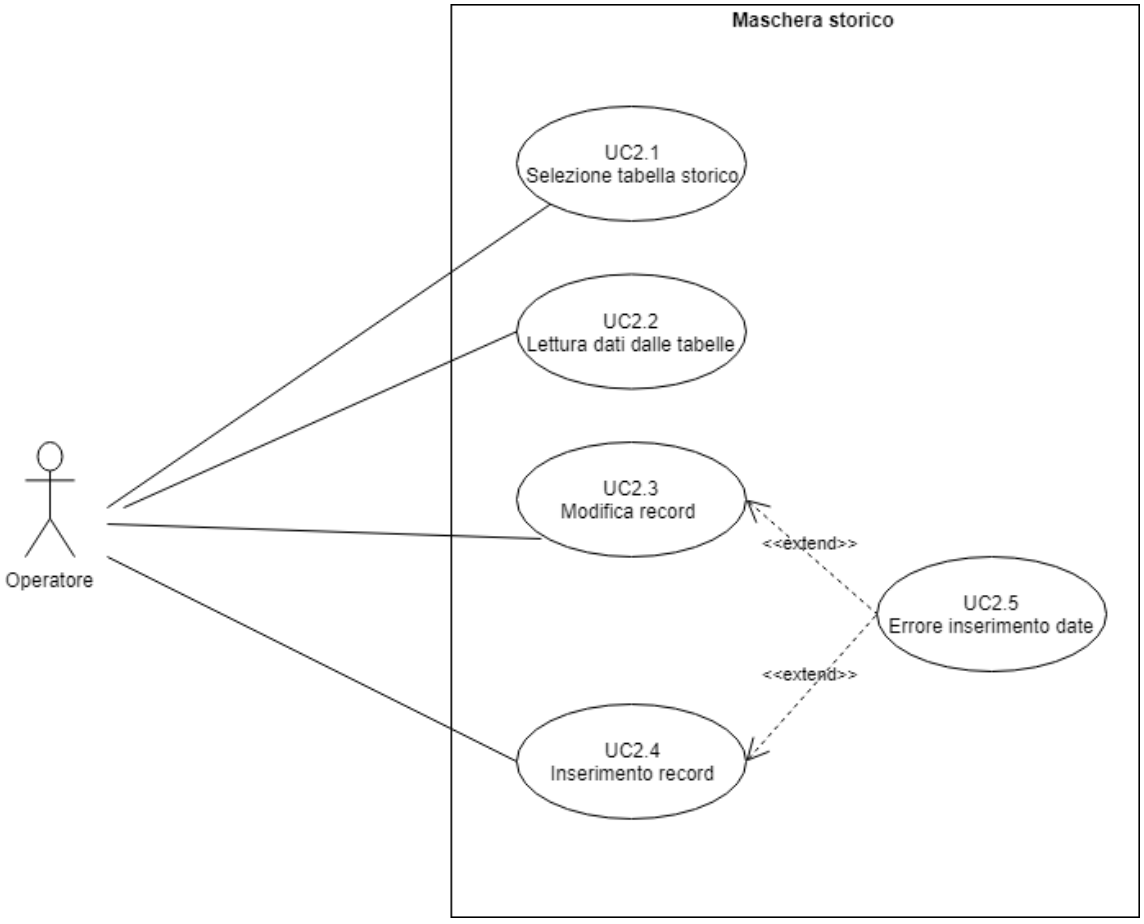

Figura 4: Diagramma dei casi d'uso della maschera contenente i dati storicizzati

3.4 KPI

In quest'ultima sezione della maschera saranno trattati i KPI (*Key Performance Indicator*). Sarà articolata in due sottosezioni: in una sarà mostrata una griglia contenente la rispettiva voce dell'indicatore in questione e ulteriori tre campi che conterranno le statistiche annuali, mensili e giornaliere, mentre nell'altra sezione un grafico che visualizzerà l'andamento dei dati corrispondente alla voce selezionata.
Il requisito finale che dovrà avere questa applicazione sarà che il mese a cui le statistiche fanno riferimento è quello precedente a quello corrente, ma siccome il progetto in questione tratta un'applicativo ancora in fase di sviluppo si farà riferimento ad un mese scelto a piacere.
Nella parte della maschera contenente il grafico che permetterà di visualizzare l'andamento dei i dati della voce selezionata, si troveranno tre bottoni, facenti riferimento alle statistiche annuali, mensili e giornaliere, ognuno dei quali permetterà di visualizzare attraverso un grafico l'andamento dei dati che fanno parte del periodo temporale di riferimento, facilitando l'analisi da parte dell'operatore.
All'atto del caricamento iniziale di questa maschera, siccome nessuna voce è stata ancora selezionata, nella sezione che contiene il grafico verrà mostrato un messaggio che invita l'operatore a selezionare un campo in modo da poter visualizzare il suo grafico e per default ogni volta che si seleziona una voce, il primo grafico che verrà mostrato sarà quello corrispondente alle statistiche annuali.
In *Figura 5* vengono mostrati schematicamente i requisiti funzionali descritti precedentemente.

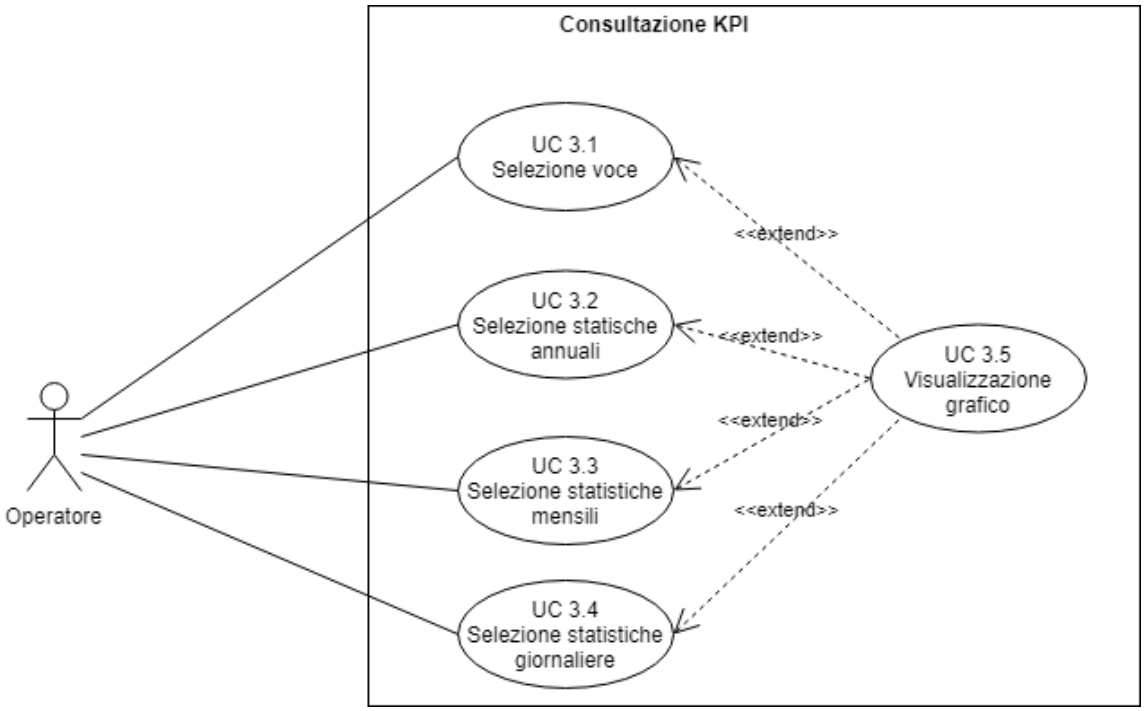

Figura 5: Diagramma dei casi d'uso della maschera contenente KPI

Tutte le relazioni che intercorrono fra queste maschere sono schematizzate in *Figura 6*:

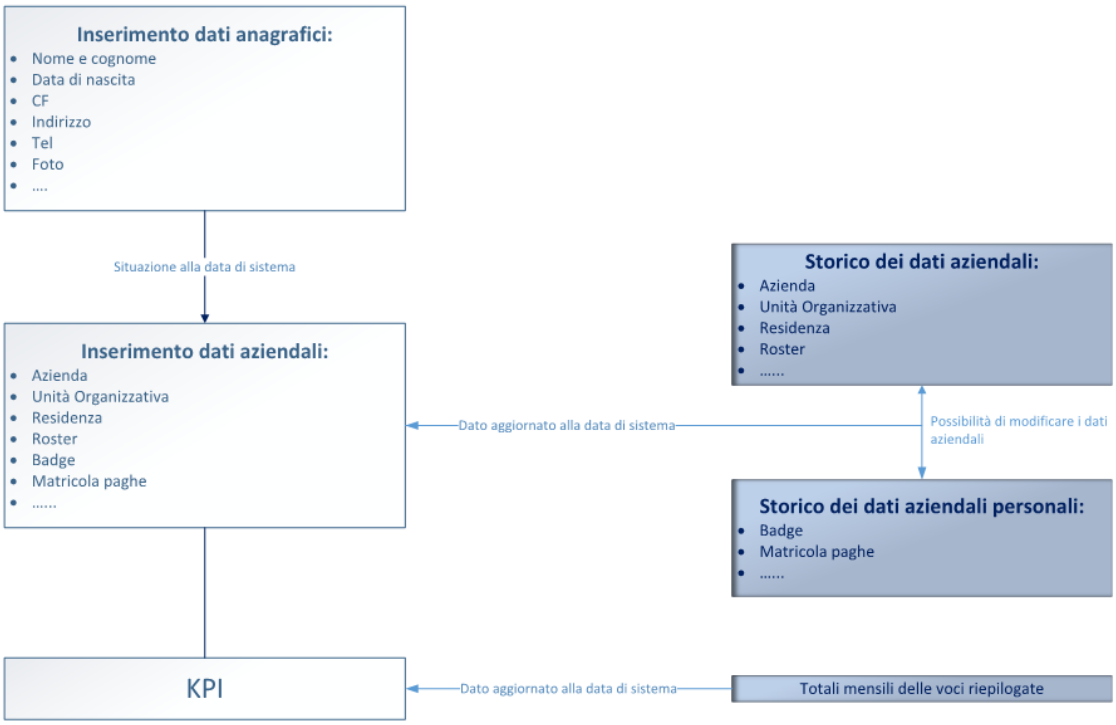

Figura 6: Interazioni tra le maschere

Facendo riferimento allo schema appena mostrato, si precisa che l'inserimento dei dati anagrafici ed aziendali fa parte della prima maschera che verrà trattata nell'applicazione. In particolar modo, i dati aziendali in questa maschera sono aggiornati alla data di sistema in quanto di sola lettura, infatti come già definito in precedenza ogni campo conterrà il valore più recente della tabella storicizzata ad esso riferito.
Sempre facendo riferimento allo schema, i due blocchi che trattano gli storici indicano che nella maschera interessata ci saranno delle tabelle che faranno riferimento a dati aziendali e personali, in entrambi i casi modificabili.
Infine la maschera con i KPI conterrà una serie di voci riepilogative: ognuna di esse mostrerà una serie di dati aggiornati alla data di sistema, che poi potranno essere visualizzati anche graficamente, come definito in precedenza.

Capitolo 4

Strumenti utilizzati

4.1 Angular

4.1.1 Introduzione

Come si evince da quanto descritto nel capitolo precedente il progetto verte sullo sviluppo di maschere lato front-end e quindi, principalmente bisogna gestire le possibili interazioni che l'operatore può avere con l'applicazione stessa.
In questo caso, è stato scelto di utilizzare il framework[1] Angular, nella sua versione 7.
Si tratta di un framework JavaScript lato client per la progettazione applicazioni web e scritto in TypeScript che rappresenta un superset del linguaggio JavaScript orientato agli oggetti che permette l'utilizzo di classi, interfacce e moduli. Si preferisce l'utilizzo di TypeScript rispetto al JavaScript in quanto è molto più compatto e permette di scrivere codice più espressivo, anche se non tutti i browser lo supportano e quindi è necessario compilare tutto il codice TypeScript sviluppato e trasformarlo in un codice che sia comprensibile agli attuali browser.

4.1.2 Angular CLI

Nella realizzazione di un nuovo progetto è estremamente importante la creazione di un ambiente di sviluppo, operazione che potrebbe risultare complicata a causa della

[1] Un *framework* è un insieme di programmi di supporto, librerie ed altri strumenti che consentono di sviluppare ed integrare le diverse componenti software di un'applicazione.

quantità di pacchetti software da integrare per la definizione di diversi aspetti che potrebbero cambiare nel tempo, come le dipendenze software.
Per semplificare l'attività di setup dell'ambiente di sviluppo, il team di Angular ha sviluppato **Angular CLI**, un ambiente a riga di comando per creare la struttura di un'applicazione Angular già configurata. La configurazione di Angular CLI necessita l'utilizzo di un package manager, cioè una collezione di strumenti che automatizzano il processo di installazione e di configurazione di un software. Nello specifico è stato utilizzato **npm**, un package manager di JavaScript basato su NodeJS che rappresenta una piattaforma open source per l'esecuzione di codice JavaScript lato server.
Da terminale Angular CLI può essere installato con il seguente comando:

```
$ npm install -g @angular/cli
```

Dopo aver creato il progetto, Angular CLI consente di ottenere un'anteprima dell'applicazione e di visuallizarla in un browser tramite il comando:

```
$ ng serve -o
```

Esso manda in esecuzione un web server che consentirà l'accesso all'applicazione all'indirizzo: ***http://localhost:4200***.
L'utilità di Angular CLI si estende anche nella possibilità di generare interi frammenti di codice, tra cui generazione di componenti, servizi, classi e altro.

4.1.3 Componenti

Essendo Angular un framework costituito da una solida architettura, uno dei concetti fondamentali su cui si basa è il concetto di *componente*, che rappresenta l'elemento costitutivo fondamentale delle applicazioni Angular. Infatti, oltre ad avere il controllo di una porzione dello schermo implementando sostanzialmente una *view*, ne visualizza i dati e agisce in base all'input dell'utente. Quindi un'applicazione Angular è un insieme di componenti che interagiscono tra di loro. In particolare, essi possono essere combinati fra di loro creando nuovi componenti organizzati in una struttura gerarchica. Un'applicazione avrà sempre una *root component* che costituisce in un certo senso il punto d'ingresso dell'applicazione stessa. Per convenzione al root component viene assegnato il nome di *AppComponent*, che naturalmente può essere cambiato a piacere.

Si può specificare che una classe rappresenta un componente attraverso il decoratore *@Component* che specifica i relativi metadati:

- **selector**: selettore che dice ad Angular di creare un'istanza di questo componente ovunque trovi il tag corrispondente nel file HTML;
- **templateUrl**: indica il file HTML che descrive il markup del componente. Esso può contenere meccanismi e direttive Angular, che alterano l'HTML in base alla logica definita nell'applicazione;
- **styleUrls**: indica l'elenco dei file CSS da applicare al markup.[3]

4.1.4 Moduli

Di notevole importanza, oltre l'AppComponent, è l'*AppModule*, che rappresenta il modulo di partenza dell'applicazione, il cosiddetto *root module*, il quale caricandosi permette l'avvio di tutti i componenti dell'applicazione.
Un'applicazione può essere composta da più moduli, ma l'importante è che sia presente il modulo di partenza, perchè senza esso non è possibile avviare un'applicazione Angular. Un *modulo* è un contenitore di funzionalità che consentono di organizzare il codice di un'applicazione.
In particolare, un modulo non è altro che una classe a cui è stato applicato un decoratore *@NgModule* avente diverse proprietà, tra cui[3]:

- **declarations**: dichiara tutti i componenti che appartengono al modulo;
- **imports**: array contenente i nomi di altri moduli, le cui classi sono impiegate nei template dei componenti dichiarati nel modulo corrente;
- **exports**: definisce quali degli elementi del modulo devono essere visibili all'esterno in modo da renderli disponibili ed utilizzarli all'interno dei template dei componenti appartenenti ad altri moduli;
- **providers**: definisce l'elenco dei provider per la registrazione dei servizi definiti nel modulo corrente;
- **bootstrap**: definisce il componente root dell'applicazione. Questa proprietà è presente solo in *AppModule*.

Ovviamente non tutte le proprietà appena descritte devono essere obbligatoriamente presenti.

4.1.5 Data binding

Senza l'utilizzo di un framework, si sarebbe responsabili di inserire i valori delle variabili dentro l'HTML e di trasformare le risposte degli utenti in azioni e aggiornamenti dei valori stessi. Scrivere manualmente ciò è però complicato e spesso soggetto ad errori.
Per questo motivo Angular supporta il *two-way data binding*, un meccanismo che coordina le varie parti di un template con le parti di un componente.[3]
La *Figura 7* mostra le possibili tipologie di data binding supportate in Angular.

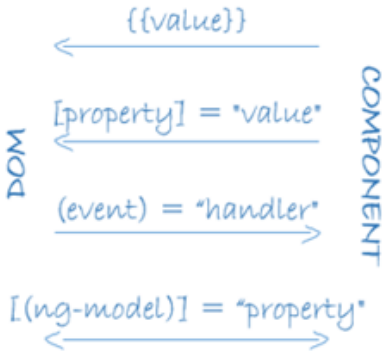

Figura 7: Tipologie di data binding

Il data binding è composto da:

- **Event binding**: aggiornamento dei dati applicativi in seguito ad azioni di input dell'utente;
- **Property binding**: interpolazione dei dati di un componente visualizzabili nella view.

Si può visualizzare l'interazione tra componenti e template mediante il data binding in *Figura 8*.

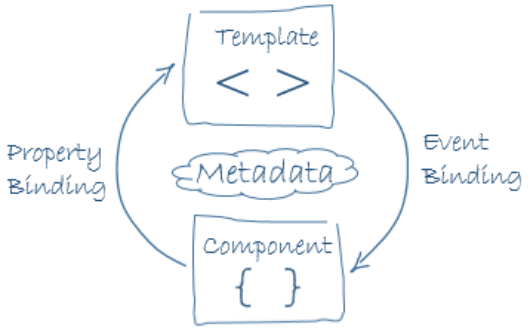

Figura 8: Interazione tra componenti e template

Invece in *Figura 9* si può notare come il *two-way data binding* sia importante anche per le comunicazioni fra un componente padre e un componente figlio.

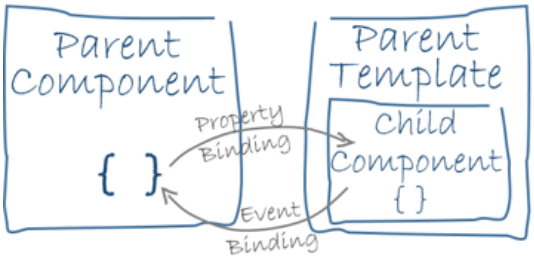

Figura 9: Interazioni tra componenti padre e figlio

Alla luce di quanto detto, possiamo rappresentare il sistema come un albero di componenti, ognuno aventi le proprie funzionalità specifiche, per una migliore separazione della logica applicativa.
Di notevole importanza è il fatto che i componenti possono comunicare tra di loro e riguardo ciò è interessante mostrare in che modo questo avviene.
Per far si che un componente visualizzi dati che vengono passati dall'esterno, ad esempio da un altro componente, e che quindi possono potenzialmente variare durante l'esecuzione dell'applicazione, si utilizza il decoratore *@Input* a fianco della dichiarazione della proprietà in cui dobbiamo andare a memorizzare questi dati.
Altro aspetto fondamentale è che l'utente interagisce con l'applicazione generando inevitabilmente eventi sugli elementi del DOM. Di fronte a questo problema, Angular consente di utilizzare una sintassi molto semplice per specificare l'evento da gestire all'interno del template HTML, infatti basta specificare il nome dell'evento

racchiuso tra parentesi tonde indicando la funzione che lo deve gestire. Inoltre, è possibile ottenere informazioni associate all'evento ed eventualmente controllarlo in maniera più accurata passando alla funzione di gestione l'oggetto *$event*.
La stessa sintassi si può usare per gestire eventi personalizzati che sono generati da un componente, infatti per generarne uno basta utilizzare il decoratore *@Output* e la classe *Event Emitter*.
Il decoratore *@Output* consente di definire una proprietà che genera un flusso di informazioni dall'interno del componente verso l'esterno, mentre la classe *Event Emitter* permette di generare un evento e di passare informazioni all'esterno, specificando il tipo di dato del payload[2] e poi invocando il metodo *emit()* della proprietà che è istanza della classe Event Emitter.

4.1.6 Ciclo di vita dei componenti

I componenti che costituiscono un'applicazione Angular vengono creati dinamicamente in base all'evoluzione dell'applicazione stessa. L'interazione con l'utente e con il server avvia la creazione dei componenti, la loro visualizzazione ed il loro aggiornamento sulla schermata e la loro distruzione. L'esistenza dei componenti durante l'esecuzione dell'applicazione attraversa diverse fasi che ne rappresentano il ciclo di vita ed Angular consente di gestirle sfruttando i cosiddetti *Lifecycle Hooks*, cioè un insieme di eventi in corrispondenza dei quali è possibile definire metodi per la loro gestione.
É opportuno evidenziare che la prima attività effettuata dal framework alla creazione di un componente è l'esecuzione del suo costruttore nel quale, siccome non sono state inizializzate le proprietà di input e non è ancora disponibile la view associata al componente stesso, non è possibile accedere agli elementi del DOM e ai componenti figli.
Durante l'esecuzione del costruttore, le fasi che si susseguono in sequenza temporale sono le seguenti[4]:

- **onChanges**: si verifica quando il valore della proprietà di input viene modificato. Oltre a verificarsi prima dell'inizializzazione del componente, si verifica anche ogni qualvolta cambia il valore delle proprietà di input;

[2] *Payload*: parte di un flusso di dati che rappresenta il contenuto informativo

- **onInit**: rappresenta la fase di inizializzazione del componente e si verifica dopo il primo evento *onChanges*. Questa fase viene eseguita una sola volta durante il ciclo di vita del componente;

- **DoCheck**: questa fase viene eseguita durante il check interno di Angular per valutare le modifiche ai componenti ed ai dati;

- **AfterContentInit**: il contenuto associato al componente è stato inizializzato, in particolare, è stato costruito l'albero degli eventuali componenti figli;

- **AfterContentChecked**: si esegue un controllo interno di Angular sui contenuti associati al componente;

- **AfterViewInit**: questa è la fase di inizializzazione della view associata al componente. In questa fase il componente risulta mappato sul DOM ed è quindi visibile;

- **AfterViewChecked**: questa fase riguarda il check interno di Angualar sulla view appena generata;

- **onDestroy**: questa è l'ultima fase del ciclo di vita di un componente e si verifica prima che Angular lo distrugga definitivamente. Questa fase viene eseguita una sola volta durante il ciclo di vita del componente.

4.1.7 Direttive

Una **direttiva** è un marcatore che estende l'HTML oppure modifica il comportamento di un elemento HTML standard e Angular definisce tre categorie di direttive. I **componenti** non sono altro che direttive con un template, cioè elementi che hanno generalmente un ruolo primario nella costruzione dell'interfaccia utente.
Poi vi sono le **direttive di attributo** che hanno il compito di modificare l'aspetto o il comportamento di un elemento. Esse si presentano come un normale attributo HTML a cui possono essere assegnate espressioni che rappresentano il comportamento desiderato.
Possiamo avere in questo caso *ngStyle* che modifica lo stile di un elemento in base al risultato della valutazione dell'espressione che gli viene assegnata oppure *ngClass*

che lavora in maniera analoga assegnando però all'elemento una classe di stile.
Infine, ci sono le **direttive strutturali** che modificano la struttura del DOM aggiungendo o rimuovendo elementi. Ad esempio, si trovano in questa categoria le direttive[5]:

- **ngIf**: aggiunge un blocco di elementi in base alla valutazione di una espressione booleana;

- **ngSwitch**: consente di generare un blocco di elementi da un insieme di possibili blocchi in base al valore di un'espressione. Si avvale della collaborazione di altre due direttive:

  - *ngSwitchCase*: serve per individuare il blocco da generare;
  - *ngSwitchDefault*: individua il blocco da generare nel caso in cui il valore dell'espressione catturata da *ngSwitch* non corrisponda a nessuno dei valori precedenti;

- **ngFor**: consente di generare un elenco di blocchi di elementi basati sul contenuto di una lista.

4.1.8 Servizi

Un **Servizio** in Angular è una classe che implementa funzionalità condivise dai vari elementi di un'applicazione, siano essi componenti oppure altri servizi.
Angular distingue i componenti dai servizi per il semplice scopo di aumentare la modularità e la riusabilità all'interno del sistema da sviluppare. Per questo motivo un componente delega ai servizi determinate attività, per esempio gestire la comunicazione con il server e reperire i dati da esso.
Alla base dei servizi vi è la **Dependency Injection**[6], che rappresenta un pattern di programmazione che fornisce come vantaggi una gestione evoluta delle dipendenze e la testabilità degli elementi di un applicazione. Grazie al sistema di dependency injection del framework un componente, un servizio o una generica classe si limita a dichiarare nel costruttore gli oggetti di cui ha bisogno, perchè sarà poi il sistema a creare le istanze necessarie e passarle al costruttore.

Per definire una classe che deve operare come un servizio si usa il decoratore *@Injectable()* che ha lo scopo di generare i metadati relativi al servizio necessario per poter effettuare una corretta dependency injection da parte del framework.
Una volta creato il servizio non è immediatamente disponibile ai componenti dell'applicazione, infatti per far ciò è necessario dichiararlo come *provider* del componente che lo utilizzerà.
La *Figura 10* rappresenta il concetto importante della dependency injection e la relazione con un componente.

Figura 10: Dependecy Injection

4.1.9 RESTful API e Servizi asincroni

Dovendo gestire una componente client avente molte richieste da inviare al server, si è reso necessario l'utilizzo di API.
Le API (*Application Program Interface*) sono un insieme di tecniche che consentono di collegarsi ad un server esterno ed eseguire alcune operazioni sui dati che esso contiene.
Per **RESTful** (**REpresentational State Transfer**), invece, si intende una serie di approcci che definiscono le regole con cui i dati sono trasmessi dal client al server. REST descrive ogni tipo di interfaccia capace di trasmettere dati per mezzo del protocollo HTTP, senza l'appoggio di tecnologie ausiliarie come cookie o protocolli per lo scambio di messaggi. Questo ha permesso la creazione della comunicazione tra client e server rendendola completamente stateless, dove ogni richiesta è svincolata dalle altre.
Le API consentono di effettuare una qualsiasi delle seguenti operazioni, denominate CRUD:

- **Create**: creazione di un elemento mediante richiesta **POST**

- **Read**: lettura di un elemento mediante richiesta **GET**
- **Update**: modifica di un elemento mediante richiesta **PUT**
- **Delete**: eliminazione di un elemento mediante richiesta **DELETE**

Queste operazioni funzionano in modo del tutto trasparente rispetto al database sul quale vengono inviate.
In Angular 7 il meccanismo di gestione asincrona del protocollo HTTP è basato sugli **Observable**[7], che rappresentano un elemento fondamentale di *Reactive Programming*, cioè un modello di programmazione per la gestione di flussi di dati asincroni. Infatti, gli Observable rappresentano questo flusso di dati (*stream*) che, nel caso delle chiamate HTTP, provengono dal server in maniera asincrona.
Questo modello di programmazione non è tipico di uno specifico linguaggio, ma può essere applicato in un qualsiasi ambiente di sviluppo e nel caso di JavaScript il suo supporto è reso disponibile da alcune librerie, tra cui **RxJS**.
Per creare una nuova istanza di Observable si utilizza il metodo **Rx.Observable.create()** che riceve come argomento una funzione (*Subscriber Function*) che viene eseguita solo nel momento in cui viene invocato il metodo **.subscribe()**. Fino a quel momento non vengono allocate risorse in memoria nè eseguite le operazioni specificate nella Subscriber Function, ma questo avviene solo quando un **Observer** si registra per ricevere le notifiche di un Observable.
Semplificando, si può considerare un *Observable* come un produttore di dati, mentre un *Observer* come un consumatore di essi.
Essi presentano tre metodi in grado di interpretare altrettanti tipi di notifiche:

- **next(value)**: metodo che deve essere sempre definito visto che viene invocato dall'oggetto Observable per consegnare un nuovo valore agli Observer;
- **error(error)**: rappresenta la funzione di callback attraverso la quale un Observer riceve le notifiche di tipo Error;
- **complete()**: rappresenta una funzione di callback da attivare al termine dell'invio delle notifiche.

É possibile annullare l'esecuzione di un Observable invocando il metodo **.unsubscribe()**.

4.2 PrimeNG

PrimeNG è una ricca collezione di componenti di Angular sviluppata da *PrimeTek Informatics*, un fornitore con anni di esperienza nello sviluppo di soluzione UI open-source. Infatti, tutti gli widgets sono open source e liberi da usare sotto la licenza MIT.[8]
Per utilizzare le componenti offerte da PrimeNG è necessario installare le dipendenze direttamente da Angular CLI:

```
$ npm install primeng - -save
```

Per utilizzare i diversi componenti grafici di PrimeNG è necessario importarli in una classe modulo da '*primeng/*'

4.3 Lodash

Lodash è una libreria JavaScript che mette a disposizione dei programmatori un'utility di funzioni utilizzate per risolvere una serie di problemi comuni attraverso la programmazione funzionale[9]. Basato su Underscore.js, Lodash ha rapidamente avuto un grande successo per la maggiore consistenza delle funzioni offerte ed il gran numero di metodi messi a disposizione per manipolare array, numeri, oggetti, stringhe, ecc.
Il vantaggio più grande di utilizzare questa libreria è che le funzioni che mette a disposizione permettono di eseguire delle operazioni complesse sulle diverse strutture dati citate precedentemente, ma con sintassi molto espressive. Tant'è che spesso, se la stessa funzione di Lodash fosse implementata interamente in codice JavaScript porterebbe alla scrittura di molte righe di codice.

Capitolo 5

Applicazione sviluppata

In questo capitolo verranno descritte puntualmente le maschere associate alle funzionalità e agli automatismi che permettono di gestire i dipendenti all'interno di un'azienda di trasporto pubblico di passeggeri , con il fine di soddisfare gli obiettivi di progetti definiti nel *Capitolo 2*.
Più precisamente verrà mostrato come sono strutturate le maschere, le funzionalità ad esse associate e le relazioni che intercorrono tra loro. Talvolta, per chiarire meglio la soluzioni di alcuni problemi affrontati, verranno mostrate alcune porzioni di codice ritenute significative.

5.1 Maschera dipendente

Ripercorrendo gli obiettivi di progetto in ordine, si parte dalla maschera del dipendente che è quella che dovrà contenere tutte le altre che caratterizzano l'applicazione, come specificato in fase di definizione dei requisiti funzionali.
La prima cosa che viene fatta all'atto del caricamento della pagina è di andare a prelevare da una delle tabelle che si trovano lato back-end le informazioni relative al dipendente, identificato da un codice. Tuttavia, per fare ciò il primo passaggio è stato di andare a definire tante classi quante erano le tabelle a cui si fa riferimento. Più precisamente si parte da una tabella **EMPLOYEE** nella quale sono presenti un insieme di attributi relativi ai dati anagrafici del dipendente e un insieme di campi che contengono degli identificatori per le altre tabelle. In particolar modo, quelle a cui si fa riferimento all'atto della chiamata iniziale sono:

- **EMPLOYEE_PROPERTY_IN_EMPLOYEE**: contiene tutti le associazioni dei dati aziendali storicizzati e le scadenze con lo status.

- **EMPLOYEE_TYPE_OF_PROPERTY** e **EMPLOYEE_PROPERTY**: contengono tutti i dati aziendali che hanno bisogno di anagrafica tipo le qualifiche, le mansioni, le scadenze, etc.

Per mantenere le stesse relazioni che intercorrono tra le tabelle, a livello di codice in quegli attributi che contengono un identificatore di un'altra tabella, è sufficiente istanziare la classe associata alla tabella di interesse.
La realizzazione della maschera del dipendente avviene definendo un componente di nome *Dipendente* che poi conterrà al suo interno tutti quelli relativi alle altre funzionalità. In particolar modo, riguardo la gestione dell'interfaccia utente, l'approccio che è stato utilizzato è quello di suddividere la finestra in due porzioni, una di esse è relativa al menu di navigazione, l'altra è riempita dinamicamente a seconda della richiesta fatta dall'operatore attraverso il click sulle varie voci del menu stesso. In ogni modo, le varie porzioni sono state associate a diversi componenti che interagiscono fra loro, in modo che si riescono a distinguere le varie funzionalità e soprattutto rendendo più efficienti eventuali operazioni di manutenzione del codice. Quindi, la struttura adottata è quella di aver un componente genitore più esterno, che definisce la maschera del dipendente, e una serie di componenti figli interni allo stesso, che vengono richiamati attraverso gli opportuni tag ad essi associati, che costituiscono le diverse sottomaschere.

Per recuperare il dipendente desiderato, all'atto del caricamento della pagina viene attivato il metodo **getEmployee()** il cui codice viene mostrato in *Figura 11*.

```
public getEmployee(idEmployee:string): Observable<EmployeeViewModelDTO> {
  const headers = new HttpHeaders().set('Content-Type', 'application/json');
  return this.http.get<EmployeeViewModelDTO>(`${this.appConfigService.urlBackEnd}/employee?idEmployee=${idEmployee}`,
    { headers: headers }).pipe(
    catchError((error) => {
      console.log('ERRORE');
      return throwError(error);
    }),
  );
}
```

Figura 11: Funzione per caricare il dipendente

Questa funzione è definita all'interno di una classe di nome *EmployeeService* che funge da servizio e viene richiamata all'interno del componente *Dipendente* nel metodo **ngOnInit()**, cioè subito dopo il termine del costruttore. Si osserva che questo metodo prende come parametro l'identificatore del dipendente e restituisce come risultato un Observable di tipo *EmployeeViewModelDTO* (che rappresenta la classe associata alla tabella *EMPLOYEE*) al quale, sempre all'interno del metodo *ngOnInit()* avviene la sottoscrizione mediante la funzione **subscribe** per poter prelevare i dati d'interesse e memorizzarli in un oggetto che mappa la tabella contenente i dati del dipendente stesso.
Inoltre, all'interno della stessa maschera si ha la funzione **save()** che provvede ad effettuare il salvataggio del dipendente. Siccome essa utilizza le informazioni provenienti sia dalla maschera contenente i dati anagrafici-aziendali che da quella contente i dati storicizzati, si ritiene più opportuno, ai fini della comprensione dello stesso metodo, descriverlo successivamente.

5.2 Menu

Prima di procedere a descrivere le funzionalità e gli automatismi che presenta questa maschera viene mostrato in che modo è stato realizzato il menù laterale di navigazione che permette di spostarsi tra le varie sottomaschere una volta cliccata la voce d'interesse.

In *Figura 12* si può osservare come appare il menù quando è selezionata la voce contenente i dati anagrafici e aziendali.

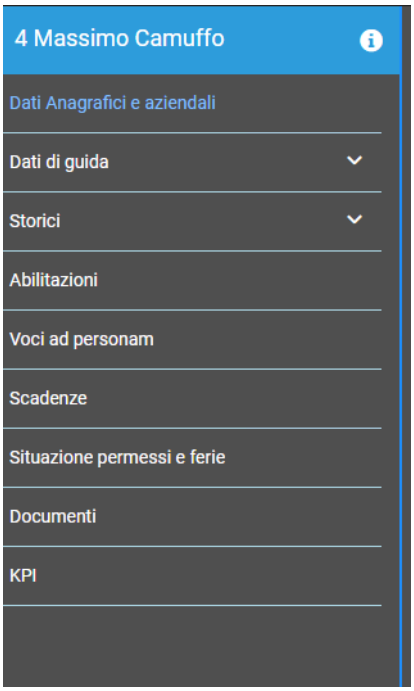

Figura 12: Menu laterale di navigazione

In questo caso, per far capire all'operatore quale maschera sta visualizzando, si evidenzia, con un colore diverso, la voce opportuna.

In *Figura 13* si può notare la comparsa del menu a tendina quando si seleziona la voce relativa agli storici.

Figura 13: Menu laterale di navigazione quando si seleziona la voce *Storici*

La realizzazione del menù è associata ad un componente di nome ***menu.component***, nel quale il meccanismo principale che deve essere implementato è il passaggio da una maschera ad un'altra. Questo avviene mediante la funzione **clickMenu()** che prende come parametro il nome della voce del menu che l'operatore ha cliccato. A questo punto la prima cosa che viene fatta è generare un evento affinchè il nome della voce selezionata del menu venga trasferito al *dipendente.component* più esterno. Questo fenomeno avviene definendo, nel componente relativo al menu, una variabile associato al decoratore @Output() e assegnandole un'istanza della classe EventEmitter. In particolare:

```
@Output() selected =new EventEmitter<any>
```

A questo punto nel metodo definito precedentemente invocando la funzione:

```
this.selected.emit(name)
```

si sta passando al componente più esterno la voce d'interesse. Questo fa si che nella sezione del template del *dipendente.component* che deve essere riempita dinamicamente, mediante la direttiva **ngSwitch*, si può valutare la voce selezionata andando a creare nel DOM solo la porzione dell'HTML d'interesse.

A questo punto nella stessa funzione del menu si procede andando a valutare se il nome della voce selezionata è uguale a "Storici", perchè in tal caso, andando a cambiare il valore di una variabile booleana utilizzata all'interno del template di *menu.component*, si decide se andare a visualizzare o meno il menu a tendina. Questo perchè quando questa porzione di menu non è visualizzata il primo click permette la relativa comparsa, altrimenti, se già visibile, un click sulla voce "Storici", permette di nasconderlo.

Inoltre ogni voce del menu a tendina viene realizzata mediante una checkbox, come si osserva in *Figura 13*, il cui valore verrà utilizzato successivamente per andare a mostrare o meno le relative tabelle nella maschera contenente i dati aziendali storicizzati.

Se invece la voce selezionata è diversa da quella degli storici allora il menu a tendina rimane nascosto.

Altra funzione che viene definita dentro *menu.component* è **sendOptionsCheckboxStorici()** che permette di inviare, mediante un servizio, un array contenente le varie opzioni del menu a tendina, selezionate attraverso le checkbox, al componente contenente i dati aziendali storicizzati e che verrà utilizzato per far comparire o meno le rispettive tabelle.

A livello di UI si riempie l'intestazione del menu con il nome, il cognome e il codice relativo al dipendente e questo avviene sfruttando il meccanismo dell'interpolazione a livello di HTML e richiamando gli opportuni attributi dell'oggetto *EmployeeViewModelDTO* valorizzato con i dati del medesima variabile che è stata precedentemente valorizzata nel *dipendente.component* attraverso la chiamata alla funzione *getEmployee()*. Il passaggio di questi valori dal componente più esterno a quello relativo al menù, avviene definendo la variabile che deve essere valorizzata con i valori d'interesse attraverso il decoratore *@Input()*.

5.3 Maschera contenente i dati anagrafici e aziendali

In questa sezione verrà mostrato in che modo è stata realizzata la maschera contenente i dati anagrafici e aziendali nel momento in cui si clicca sulla voce corrispondente del menu, descrivendo poi puntualmente tutti gli automatismi che sono ad essa associati.

In *Figura 14* si può osservare come appare la maschera.

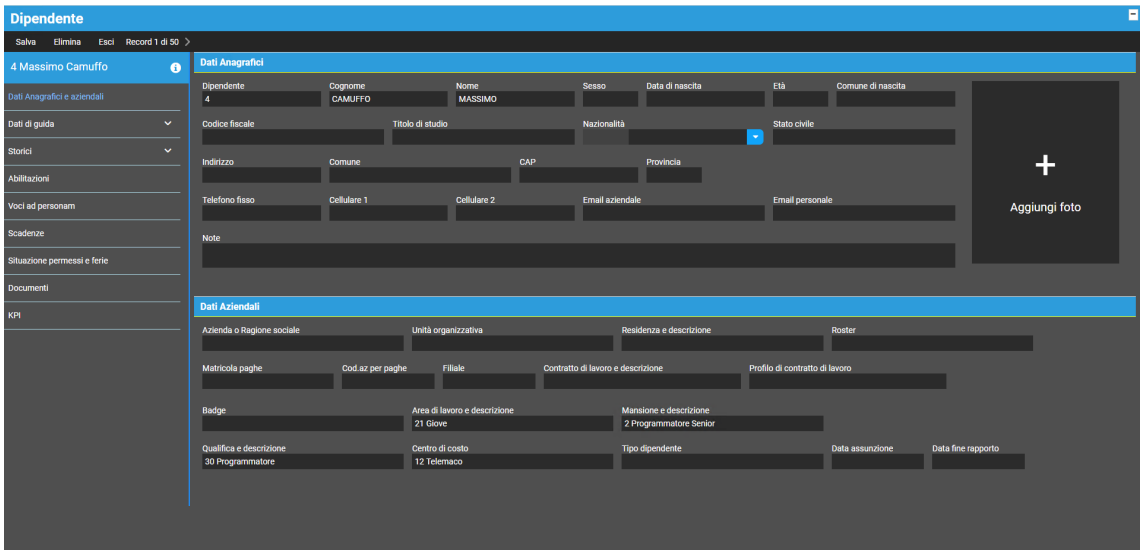

Figura 14: Maschera contenente i dati anagrafici e aziendali

Si osserva che a livello di interfaccia utente si è deciso di suddividere la maschera in due sezioni, poste una sotto l'altra, contenenti rispettivamente una form riferita ai dati anagrafici ed una riferita a quelli aziendali.

Per rendere ancora più modulare l'applicazione sono stati definiti due componenti riferiti proprio alle due sezioni che compaiono, dato che le interazioni che si possono avere con esse sono differenti.

Infatti, la maschera contenente i dati anagrafici, oltre a mostrare i valori attuali, permette all'operatore di modificare liberamente ognuno di essi, mentre quella contenente i dati aziendali si presenta come una form con tutti i campi che sono di sola lettura.

Ricapitolando, sono stati definiti tre componenti:

- ***datiDipendente.component***: racchiude l'intera maschera contenente i dati anagrafici ed aziendali;
- ***datiAnagrafici.component***: racchiude la sezione che fa riferimento ai soli dati anagrafici del dipendente;
- ***datiAziendali.component***: racchiude la sezione riguardante i dati aziendali del dipendente.

Per evidenziare il tutto, visto che poi le relazioni che intercorrono fra queste maschere sono fondamentali ai fini del funzionamento dell'applicazione stessa, in *Figura 15*, vengono mostrati graficamente i tre componenti sopra citati.

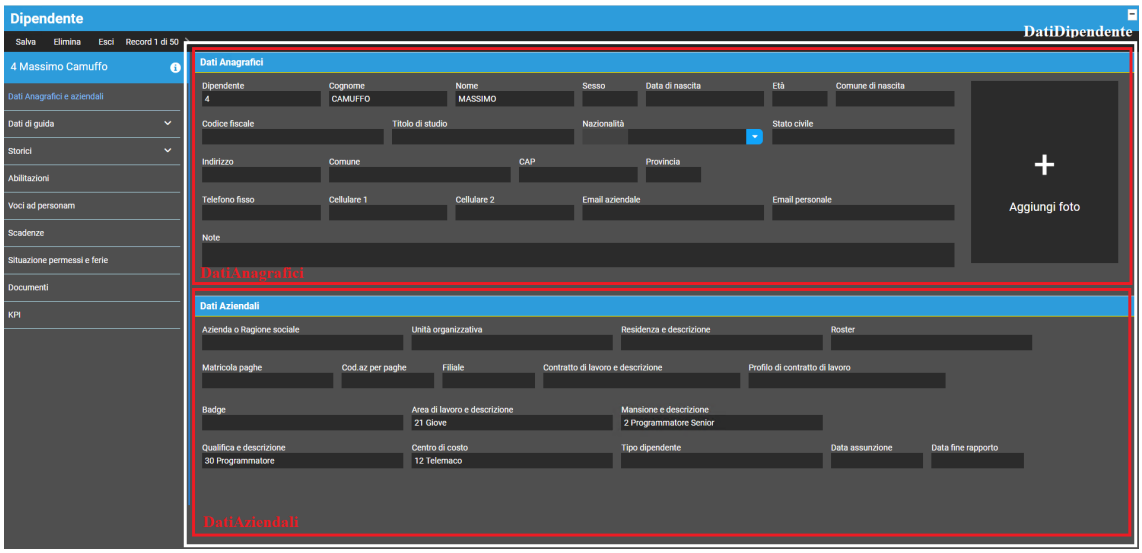

Figura 15: Suddivisione delle componenti

Nel template del componente ***datiDipendente.component*** viene applicata la suddivisione della maschera nelle due sezioni verticali ed ognuna di esse viene riempita con il template del componente interessato, specificando il tag ad esso associato.
In questo caso il meccanismo che si deve instaurare è quello di andare a filtrare da tutti i dati del dipendente che all'atto del caricamento della applicazione sono stati acquisiti solo quelli che sono relativi alla sua anagrafica e ai suoi dati aziendali.
Per fare ciò, inizialmente è stata definita in *datiDipendente.component* una variabile mediante il decoratore @Input() la quale viene valorizzata attraverso una direttiva HTML nel componente *dipendente.component* con tutti i dati del dipendente che sono stati prelevati mediante la precendente chiamata alla funzione *getEmployee()*.

Successivamente, nel metodo **ngOnInit()** del componente *datiDipendente.component* si vanno a filtrare, mediante la funzione ***.filter***, applicata all'oggetto contenente i dati del dipendente, solo quelli che ci interessano per la maschera che si sta descrivendo.

Fatto ciò questi dati vengono ulteriormente divisi in quelli che trattano esclusivamente i dati anagrafici e quelli aziendali, salvandoli in due variabili distinte, e poi con un meccanismo analogo a quanto detto prima queste due variabili vengono passate ai componenti figli ***datiAnagrafici.component*** e ***datiAziendali.component***.

In realtà, riguardo quest'ultimi c'è da fare una precisazione, ovvero il meccanismo con cui vengono prelevati non è semplice quanto quello per ottenere i dati esclusivamente anagrafici. Infatti, come già definito nei requisiti funzionali dell'applicazione, la sezione che contiene i dati aziendali presenta una serie di campi di input di sola lettura che devono essere valorizzati con i dati più recenti della relativa tabella contenuta nella sezione degli storici.

Per fare questo sono state utilizzate delle funzioni della libreria *Lodash* che hanno permesso di gestire ciò in modo molto semplice.

Prima di passare al commento del relativo codice, vengono mostrate le istruzioni di interesse in *Figura 16*.

```
let array=_.values(_.groupBy(this.employeePropertyInEmployee, 'employeeProperty.employeeTypeOfProperty.employeeTypeOfPropertyCode'));
for(let i=0; i<array.length; i++) {
    let first= _.head(_.orderBy(array[i],(r)=>r.validityEnd, ['desc']))
    switch(first.employeeProperty.employeeTypeOfProperty.employeeTypeOfPropertyCode) {
      case 'ADL': this.employeeLatestProperty.workDesc=first.employeeProperty.code+" "+first.employeeProperty.desc;break;
      case 'QUAL': this.employeeLatestProperty.qualificationDesc=first.employeeProperty.code+" "+first.employeeProperty.desc;break;
      case 'MAN': this.employeeLatestProperty.jobDesc=first.employeeProperty.code+" "+first.employeeProperty.desc;break;
      case 'CDC': this.employeeLatestProperty.costCenterDesc=first.employeeProperty.code+" "+first.employeeProperty.desc;break;
      case 'LIV': this.employeeLatestProperty.levelDesc=first.employeeProperty.code+" "+first.employeeProperty.desc;break;
    }
}
let firstCrewBaseStation=_.head(_.orderBy(this.employeeAnagraficaAziendale.crewBaseStationInEmployee,(r)=>r.validityEnd, ['desc']))
this.employeeLatestProperty.crewBaseStationDesc=firstCrewBaseStation.idCrewBaseStation+ " "+firstCrewBaseStation.crewBaseStation.crewBaseStationDescription;
}
```

Figura 16: Funzione che filtra i dati aziendali più recenti di ogni campo

Le funzioni Lodash che sono state utilizzate sono[10]:

- **_.values**: crea un array con i valori delle proprietà dell'oggetto che viene passato come parametro;
- **_.groupBy**: crea un oggetto composto dalle chiavi generate dai risultati dell'esecuzione su ciascun elemento di ciò che viene passato come secondo parametro. Il valore corrispondente ad ogni chiave è un array contenente gli elementi responsabili della generazione della chiave stessa;

- **_.orderBy**: crea un array di elementi ordinati secondo il valore che viene passato come secondo parametro;
- **_.head**: estrae il primo elemento dell'array.

Il metodo ***_.groupBy*** viene applicato all'oggetto di nome *employeePropertyInEmployee* che contiene tutte le proprietà riguardanti i dati aziendali di un dipendente e il campo rispetto cui questi valori devono essere raggruppati è la proprietà *employeeTypeOfPropertyCode* che rappresenta un codice identificativo per ogni differente dato aziendale.
Quindi come già detto prima, il risultato che si ottiene è un oggetto che presenta tante chiavi quanti sono i valori di *employeeTypeOfPropertyCode* differenti e per ognuno di essi si hanno gli elementi ad esso associati.
Successivamente a questo risultato viene applicato il metodo ***_.values*** che estrae i valori delle chiavi che sono stati precedentemente raggruppati e li inserisce all'interno di un array di nome *array*.
A questo punto viene definito un ciclo che itera un numero di volte pari alla dimensione dell'array appena definito e ad ogni iterazione applica su ogni singolo elemento il metodo ***_.orderBy*** che permette di ordinare gli elementi, presenti dentro ogni indice dell'array, in modo decrescente rispetto l'attributo *validityEnd* che rappresenta la data di fine. Per ogni valore estratto si valuta il codice a cui è associato e in base a ciò si va a popolare una struttura dati che contiene tante proprietà quanti sono i campi di input della form contenente i dati aziendali.
In maniera analoga le ultime due righe di codice permettono di estrarre la residenza più recente del dipendente e popolare poi il campo della struttura dati che verrà trasferita al componente ***datiAziendali.component***.
A questo punto in ***datiAnagrafici.component*** si definisce una variabile con la direttiva @Input() valorizzata con i dati anagrafici che sono stati precedentemente estratti e i vari campi di input del template vengono inizialmente riempiti con i valori delle proprietà di questo oggetto mediante un meccanismo di interpolazione.
Siccome in questa sezione è possibile modificare ogni campo a piacimento è necessario definire un meccanismo di salvataggio che, in maniera analoga a quanto detto circa *dipendente.component*, verrà definito in seguito.
Infine, in modo simile a quanto detto prima per far si che i campi della form dei

dati aziendali vengano popolati con i valori più recenti attraverso un meccanismo di binding fra *datiDipendente.component* e *datiAziendali.component* si trasferiscono i valori che sono stati filtrati dal componente padre al figlio. Tuttavia, siccome il template di quest'ultimo è caratterizzato da un insieme di campi di input di sola lettura, rispetto a quanto detto prima non si deve implementare nessun meccanismo di salvataggio.

5.4 Maschera contenente i dati storicizzati

Dopo aver descritto in modo puntuale le funzionalità della maschera contenente i dati anagrafici e aziendali si procede andando a spiegare tutte quelle che sono le caratteristiche della maschera contenente i dati storicizzati. Essa è collegata con quanto detto nella sezione precedente, infatti se si volesse andare a cambiare un dato aziendale non si potrebbe agire direttamente nella maschera d'interesse, perchè tale campo di input è di sola lettura, tuttavia per far comparire un valore diverso si deve agire direttamente nella maschera contenente i dati storicizzati.
A questo punto, con il fine di rendere più chiare le funzionalità che tra breve verranno esposte, si mostra in *Figura 17* come questa maschera compare all'atto della selezione dell'opportuna voce nel menu laterale di navigazione.

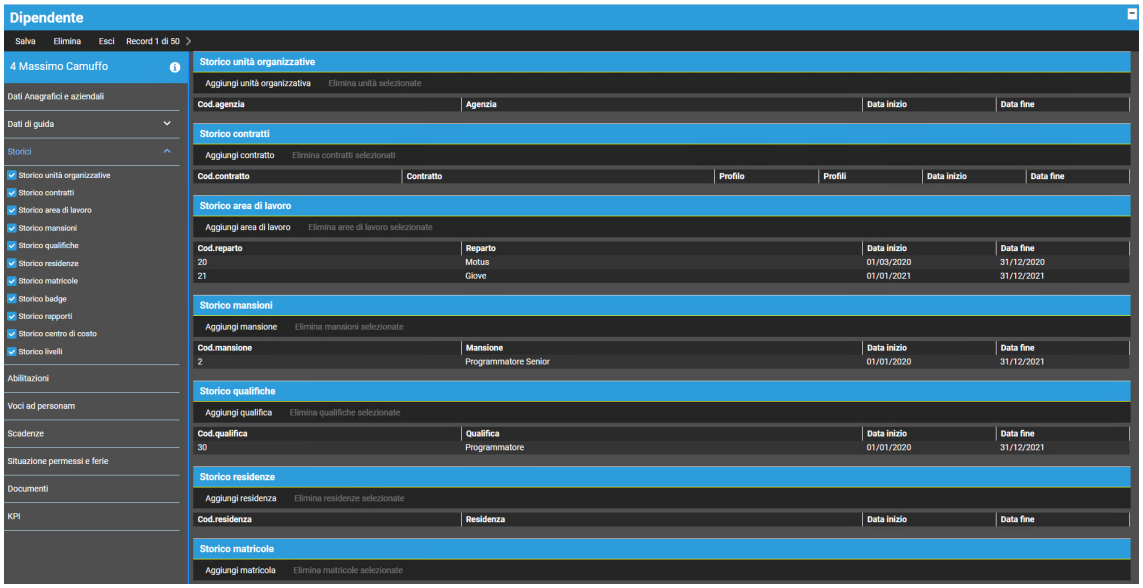

Figura 17: Maschera che mostra i dati storicizzati all'atto del caricamento della stessa

Si osserva che una volta che si è cliccata la voce per far comparire la maschera nel menu laterale di navigazione ne compare uno nuovo, che alla selezione delle altri voci non è possibile visualizzare, nel quale si hanno una serie di checkbox che sono tutte già selezionate. Ognuna di esse è associata ad una tabella nella maschera dei dati storicizzati e la deselezione di una di tali checkbox determina la scomparsa della relativa tabella.
Per chiarire meglio questo aspetto in *Figura 18* si mostra la stessa maschera dove però sono selezionate solo le voci riferite allo storico dell'area di lavoro, delle mansioni e delle qualifiche nella quale si nota la comparsa di sole tre tabelle.

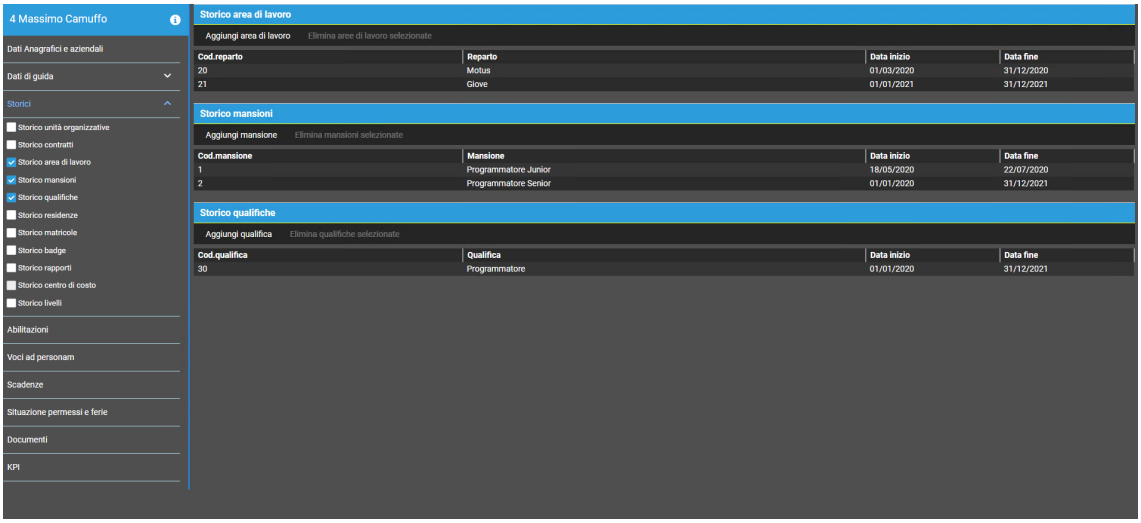

Figura 18: Maschera dello storico con solo alcune voci che sono state selezionate

Quello che si potrebbe rischiare in questo caso è che l'operatore vada a deselezionare tutte le voci e quindi, per gestire questa situazione, si è deciso di far comparire un messaggio che invita l'utente a selezionarne una per poter poi osservarne i dati. Questa situazione particolare viene mostrato in *Figura 19*.

Figura 19: Maschera dello storico con nessuna voce selezionata

A questo punto, dopo che l'operatore che utilizza l'applicazione si trova davanti questa maschera può decidere di andare a modificare le tabelle esistenti, come già descritto nei requisiti funzionali dell'applicazione.
In particolar modo, egli può decidere di andare ad aggiungere una nuova riga in ogni tabella e questo avviene cliccando l'opportuno pulsante posto nella toolbar di ogni tabella, oppure può decidere di modificare un record già esistente semplicemente cliccando due volte sulla riga di interesse.
In entrambi i casi compare un wizard nel quale l'operatore può compiere l'operazione desiderata, che però presenta caratteristiche diverse a seconda di ciò che l'utente deve svolgere.
In *Figura 20* si può osservare la schermata che compare nel momento in cui l'utente clicca al di sopra di una riga per andarla a modificare.

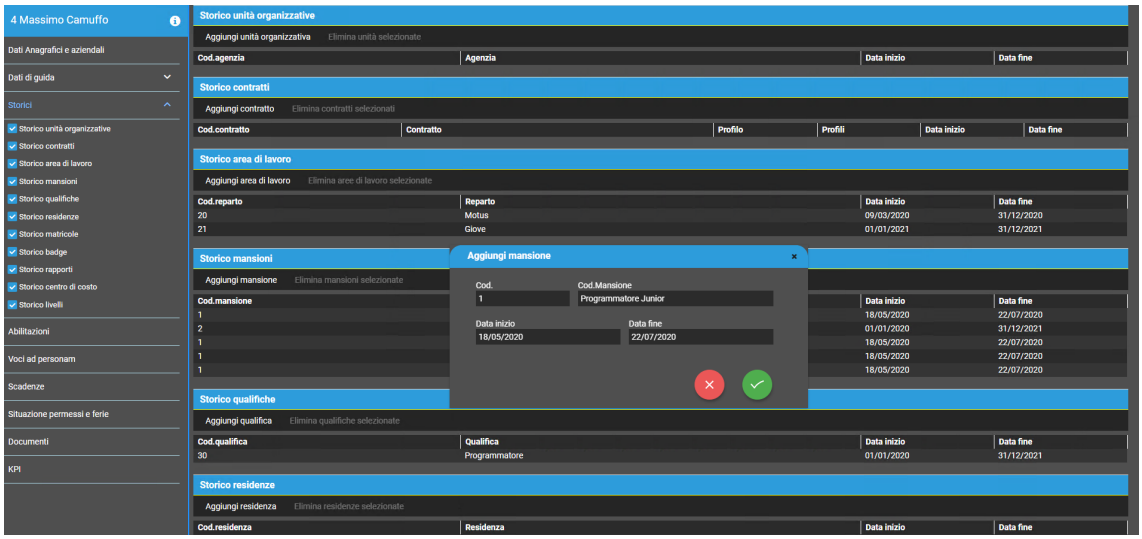

Figura 20: Schermata per la modifica di una riga già esistente

Si osserva che il wizard che compare è composto da 4 campi, denominati allo stesso modo rispetto le colonne della tabella a cui si riferiscono e che, nel caso di un'operazione di modifica, vengono già riempiti con i valori del record stesso. Questi campi non sono tutti modificabili e questo perchè l'operatore può decidere di cambiare solamente la data inizio e la data fine, mentre i campi che fanno riferimento al campo del codice e della descrizione sono di sola lettura. Una nota che c'è da fare in questo caso è che, come si può notare dalla *Figura 17*, alcune tabelle non presentano solo quattro campi, come ad esempio quella contenente i dati storicizzati relativi ai contratti. In questo caso, ovviamente nel wizard che permette di andare a modificare i valori, o eventualmente inserirne dei nuovi, dovranno comparire più campi, anche se ai fini di questo progetto queste tipologie di tabelle non sono state prese in considerazione.

Invece se l'utente decide di andare ad inserire una nuova riga all'interno di una tabella il wizard che compare è leggermente differente, come si può osservare in *Figura 21*.

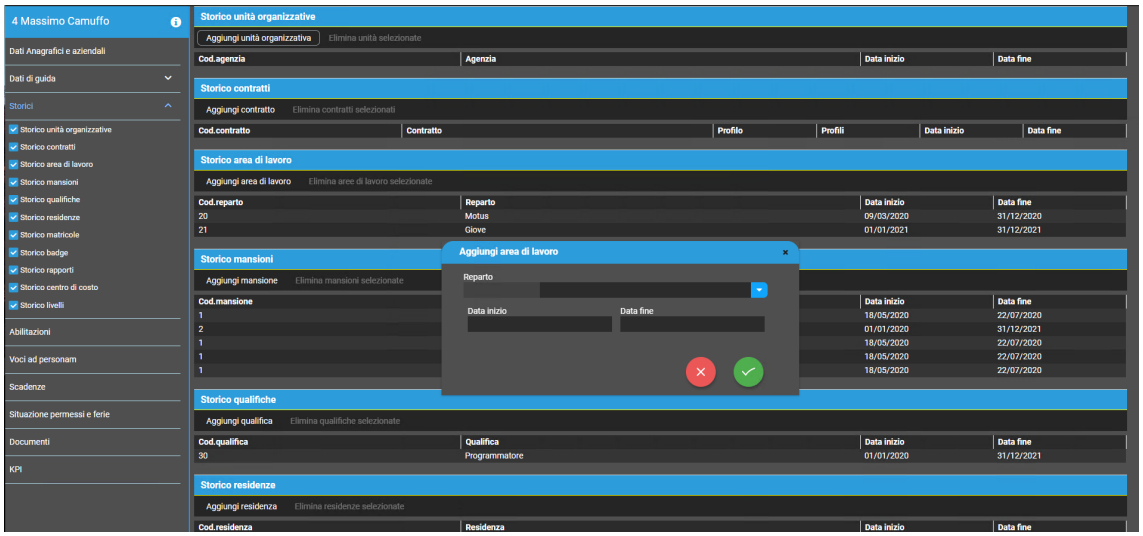

Figura 21: Schermata per l'inserimento di una nuova riga

Dopo aver presentato quello che succede a seconda delle varie azioni che l'utente può andare ad effettuare si può procedere descrivendo puntualmente le funzionalità che sono associate a questa maschera e quali sono gli automatismi.

In primo luogo, a livello di template sono state definite un numero di tabelle pari alle voci del menu a tendina utilizzando il componente **p-table** della libreria *PrimeNg*. Come si può osservare dalla documentazione del componente stesso, esso presenta l'attributo *value* che permette di specificare la struttura dati da associare al componente con cui andare a popolare poi le varie celle. Infatti ogni tabella viene associata ad un oggetto che viene valorizzato all'interno del metodo **ngOnInit()** del componente stesso nel quale si vanno a filtrare dai dati del dipendente solo quelli appartenenti alla tabella direttamente interessata. Per facilitarne la comprensione si riporta in *Figura 22* il codice di un solo caso, in quanto gli altri sono analoghi.

```
this.employeeFilterFieldAreaDiLavoro = this.employeeStorico.employeePropertyInEmployee.filter(
  c => c.employeeProperty != null && c.employeeProperty.employeeTypeOfProperty != null &&
  c.employeeProperty.employeeTypeOfProperty.employeeTypeOfPropertyCode == "ADL");
```

Figura 22: Codice per l'estrazione dei valori di interesse dai dati del dipendente

Si osserva che si filtra dall'oggetto *employeeStorico* che viene popolato all'atto della chiamata stessa alla maschera con l'oggetto contenente tutti i dati del dipendente presente in *dipendente.component*, quindi anche in questo caso l'oggetto *employeeStorico* definito all'interno di *storici.component* è associato al decoratore *@Input()*. Nel filtraggio si definiscono una serie di condizioni booleane tra cui quella che permette di estrarre i dati associati alla tabella di interesse sulla base della proprietà *employeeTypeOfPropertyCode* che rappresenta un codice con cui si identificano le varie tipologie di dati aziendali.
Inoltre, sempre nel metodo **ngOnInit** si procede a selezionare tutte le checkbox per far comparire tutte le tabelle, visto che come già indicato nel componente relativo al menu, all'atto del caricamento della maschera degli storici tutte le voci sono già selezionate. Per fare ciò, è stato definito un array dove ogni elemento presenta un attributo *value* valorizzato con un nome caratterizzante per la tabella stessa, e un attributo *selected* attualmente valorizzato a *true*.
Per andare a gestire dinamicamente la generazione delle tabelle è necessario che ogni volta che si seleziona o si deseleziona una voce dal menu a tendina l'array contenente le opzioni scelte venga passato a *storici.component* in modo tale che poi mediante una serie di direttive **ngIf* si va a controllare se generare la tabella oppure no. Visto che si deve effettuare un passaggio di una variabile da un componente ad un altro dove però entrambi sono allo stesso livello, non si ricade in una delle situazioni già viste in cui si passa una variabile da un componente padre ad un componente figlio mediante l'annotazione *@Input()*. A questo scopo è stato realizzato un servizio apposito. Esso è stato chiamato **EmployeeState** e, come già definito nel capitolo che tratta gli strumenti utilizzati, presenta prima della definizione della classe stessa l'annotazione *@Injectable()*. All'interno di questa classe è stato definito un *Observable* che poi viene utilizzato per trasferire le voci selezionate fra i due componenti interessati. La *Figura 23* mostra il codice e il metodo nel servizio *EmployeeState* utilizzati per realizzare tale funzionalità.

```
// Observable string source
private OptionToStoriciSource = new Subject<string[]>();

// Observable string stream
OptionToStorici$ = this.OptionToStoriciSource.asObservable();

public sendOptionToStorici(selectedOptions: string[]) {
  this.OptionToStoriciSource.next(selectedOptions);
}
```

Figura 23: Definizione Observable per trasferire le opzioni al componente Storici

Nella prima riga di codice sfruttando la libreria *RxJS* si definisce una variabile di tipo $Subject < string[] >$.[1] Successivamente si applica a questa variabile il metodo ***asObservable()*** che permette di creare un osservabile ed infine si definisce la funzione ***SendOptionToStorici*** che prende come parametro le voci selezionate e mediante il metodo ***next()*** emette questi valori al componente che ha effettuato la sottoscrizione all'osservabile in questione.
A questo punto nel componente del menu, nelle varie voci interessate, si associa l'evento *onChange* ad una funzione (chiamata ***sendOptionsCheckboxStorici()***) la quale attiva il metodo associato al servizio precedente, passandogli come parametro un array contenente le voci selezionate, che permette di inviare tale struttura dati al componente ***storici.component***. Ovviamente solo questo non basta in quanto per far si che tale flusso di dati arrivi a destinazione è necessario che il componente ***storici.component*** effettui la sottoscrizione all'osservabile d'interesse, mediante il metodo ***subscribe()*** in modo tale che ogni volta che viene selezionata o meno una voce nel menù a tendina il componente contenente gli storici riceva sempre i dati aggiornati. Tutto questo meccanismo viene mostrato in *Figura 24* dove si osserva che all'interno del metodo *subscribe()* sono definite le istruzioni per far comparire o meno le tabelle.

[1] **Subject** rappresenta un tipo di dato che combina contemporaneamente le funzionalità di un Observable e di un Observer.

```
this.employeestate.OptionToStorici$.subscribe(option => {
  this.showMessageStorici = false;
  if (option.length == 0) this.showMessageStorici = true;
  for (let i = 0; i < this.showOptionsStorici.length; i++) {
    this.showOptionsStorici[i].selected = false;
  }
  for (let i = 0; i < this.showOptionsStorici.length; i++) {
    for (let j = 0; j < option.length; j++) {
      if (this.showOptionsStorici[i].value == option[j]) {
        this.showOptionsStorici[i].selected = true;
      }
    }
  }
})
```

Figura 24: Sottoscrizione del componente storici e gestione della comparsa/scomparsa delle tabelle

Se le opzioni che vengono trasferite a questo componente sono zero allora viene valorizzato la variabile *showMessageStorici* a true e questo permette di visualizzare la schermata come già mostrato in *Figura 19*. Successivamente si ha un ciclo in cui tutte le tabelle vengono fatte scomparire e questo serve per far si che all'atto dell'invio delle voci selezionate, quelle visualizzate in precedenza non abbiano alcun effetto. Infine, si definisce un ciclo innestato in un altro ciclo dove ad ogni iterazione di quello esterno si confronta il nome simbolico associato ad ogni tabella con tutte le voci presenti dentro l'array delle opzioni in modo da valutare se visualizzare o meno la tabella in questione.

A questo punto, dopo aver capito come avviene il meccanismo che permette di mostrare e nascondere le tabelle dalla griglia, si può procedere andando a vedere come avvengono i meccanismi di modifica ed inserimento di un record.

Si decide di partire dalla modifica di una riga già esistente che, come già detto, avviene con un doppio click sul record in questione e permette di far comparire un wizard dove si possono modificare solo i valori associati ai campi della data iniziale e finale. A livello di template si è utilizzato il componente **p-dialog** di *PrimeNg* che permette di andare a visualizzare una finestra che si sovrappone a quella corrente. Ovviamente non si è realizzata una finestra per ognuna delle tabelle, ma ne è stata realizzata solamente una e per andarla a riempire correttamente è stata associata ad essa una struttura dati che presenta cinque campi: uno di questi si riferisce all'id del dato aziendale che si sta andando a modificare e gli altri quattro campi fanno riferimento al codice, alla descrizione, alla data inizio e alla data fine, campi comuni a tutte le tabelle che servono per andare a cambiare correttamente il valore della label

associata al campo input a seconda della tabella che si vuole modificare. All'atto del doppio click sulla riga viene eseguito il metodo ***showPopupToEdit()*** al quale vengono passati un numero di parametri tali da andare a riempire correttamente i valori della struttura dati descritta precedentemente. Inoltre, per far si che i campi di input siano già riempiti con i valori della riga di interesse, si è istanziata la classe *CustomEditClass*, che presenta un numero di proprietà pari a quelle della struttura dati descritta precedentemente, che però vengono valorizzate con i dati della riga che si è selezionata, come viene mostrato in *Figura 20*. Si osserva che questa finestra presenta due bottoni, rispettivamente per l'annullamento e la conferma dell'opzione. Il primo fa sì che scompaia il wizard, mentre il secondo risulta più complesso. Infatti, come prima cosa viene fatto un controllo sulle date che sono state inserite: se la data di inizio è più piccola di quella finale, oppure almeno uno dei campi è vuoto, compare un pop-up nel quale viene segnalato all'operatore che tipologia di errore ha commesso e il pulsante non ha alcun effetto. Se invece l'inserimento dei dati è andato a buon fine allora si controlla la tipologia della tabella che si sta andando a modificare mediante un costrutto *switch-case*. In *Figura 25* si mostra una porzione di codice che mostra le varie funzioni che vengono chiamate.

```
case 'mansione': {
  this.employeeFilterFieldMansioni.filter(c => {
    this.confirmEmployeeStoriciChange(c);
  });
  this.updateEmployeeStorico(this.employeeFilterFieldMansioni);
}; break;
```

Figura 25: Modifica di un record appartenente alla tabella delle "Mansioni"

A seconda della tabella su cui si sta operando, si filtra l'oggetto utilizzato per andare a popolare le varie celle in modo tale che ad ogni riga viene applicata la funzione ***confirmEmployeeStoriciChange()*** che, dopo aver valutato che si tratta di un'operazione di modifica controllando il valore di una variabile booleana apposita, procede andando a controllare se il codice associato alla riga della tabella passata come parametro alla funzione corrisponde con quello della riga appena modificata. In caso positivo procede andando ad aggiornare i contenuti dei campi relativi alle date, mentre in caso negativo non succede nulla e si passa a valutare la successiva riga della tabella.

L'ultima istruzione è l'esecuzione della funzione ***updateEmployeeStorico*** che pren-

de come parametro la struttura dati con la quale si sono popolate le tabelle d'interesse e procede andando ad aggiornare l'oggetto che è stato passato all'atto del caricamento della maschera contenente tutti i dati del dipendente. Questa operazione sarà necessaria per mantenere le modifiche in modo permanente quando nella prossima sezione verrà definito il metodo che provvede a salvare la maschera.

Riguardo l'inserimento di un nuovo record in una tabella qualsiasi la situazione è simile a quella che è stata appena descritta. Infatti, si utilizza un wizard identico, tranne per il fatto che i due campi che prima erano di sola lettura, vengono sostituiti da un campo unico da cui è possibile scegliere da un insieme di valori predefiniti. Inizialmente, tutti i campi sono vuoti dato che si tratta di un'operazione di inserimento, come è già stato mostrato in *Figura 21*. Il processo diverso rispetto al precedente che si è dovuto gestire è stato quello di dover caricare fra l'insieme dei valori predefiniti da cui l'utente può andare scegliere solo quelli associati alla tabella nella quale si sta inserendo una nuova riga.
Innanzitutto, è stato necessario implementare una funzione che andasse a prelevare tutte le opzioni possibili e per fare ciò è stato definito il servizio **EmployeePropertyService** che presenta un unico metodo ***getAll()*** che permette di estrarre tutte le possibili opzioni e il cui codice è mostrato in *Figura 26*.

```
@Injectable()
export class EmployeePropertyService {
  constructor(private http: HttpClient, private appConfigService: AppConfigService) {
  }

  public getAll(): Observable<EmployeePropertyViewModelDTO[]> {
    const headers = new HttpHeaders().set('Content-Type', 'application/json');
    return this.http.get<EmployeePropertyViewModelDTO[]>(`${this.appConfigService.urlBackEnd}/employeeproperty/list`,
      { headers: headers }).pipe(
      catchError((error) => {
        console.log('ERRORE');
        return throwError(error);
      }),
    );
  }
}
```

Figura 26: Funzioni per estrarre tutte le possibili voci per l'inserimento di una riga

Si osserva che questo metodo è molto simile a quello iniziale per estrarre il dipendente di interesse, tranne per il fatto che cambia il tipo dell'Observable e cambia il path nella chiamata GET che viene effettuata.
A questo punto nel costruttore del componente ***storici.component*** si istanzia un oggetto di tipo *EmployeePropertyService* e si chiama questa funzione, andando a memorizzare il risultato all'interno di una variabile. Successivamente, nel momento in cui si decide di attivare la funzione di inserimento cliccando nell'opportuno bottone della toolbar della tabella, si va a filtrare questo oggetto sulla base del codice del dato aziendale contenuto nella tabella nella quale si sta andando ad aggiungere una nuova riga. In questo modo, tra tutti i valori che inizialmente sono stati estratti mediante la chiamata ***getAll()***, si acquisiscono solamente quelli di interesse in funzione della tabella nella quale si sta andando ad inserire la nuova riga. Una volta che sono stati inseriti i valori associati alla riga, all'atto della conferma, si attiva il metodo sul controllo delle date che funziona in maniera analoga a prima e, solo dopo che il controllo è andato a buon fine si attiva la parte di codice che permette di aggiornare la tabella. Dopo aver selezionato quella nella quale si sta andando ad inserire il dato, si attiva sempre il metodo ***confirmEmployeeStoriciChange()*** che questa volta prende come parametro l'oggetto che è stato utilizzato in fase di inizializzazione della maschera per popolare la tabella nella quale si sta effettuando l'operazione. A questo punto, dopo aver verificato che si tratta di un'operazione di inserimento, procede andando ad istanziare la classe *EmployeePropertyInEmployeeViewModelDTO* che rappresenta quella classe associata ad ogni riga. Successivamente valorizza i campi dell'oggetto appena creato con quelle che l'utente ha inserito e poi procede effettuando due operazioni di ***push***. La prima viene applicata all'oggetto che è stato passato come parametro e permette, dopo aver confermato l'inserimento, di vedere che si è aggiunta una nuova riga nella tabella, mentre la seconda viene eseguita sull'oggetto *employeeStorico*, che è quello contenente tutti i dati del dipendente. Quest'ultima operazione è necessaria per implementare il meccanismo di salvataggio che verrà descritto fra poco.

5.5 Salvataggio

In questa sezione verrà mostrato il meccanismo che si attiva quando si vuole andare a salvare il contenuto della maschera. Questo processo è necessario in quanto sia nella maschera contenente i dati anagrafici/aziendali che in quella contenente gli storici, l'operatore può effettuare delle operazioni di modifica o inserimento e quindi senza questo meccanismo, all'atto della ricarica della pagina tutti i dati verrebbero persi. Infatti, per garantire la permanenza dei dati, è necessario andare a modificare le tabelle lato back-end e questo perchè i dati a cui l'applicazione fa riferimento non sono creati al runtime dell'applicazione stessa, bensì sono memorizzati in un database.

Il pulsante che permette di andare ad effettuare questa operazione è posto in alto a sinistra nel template del componente ***dipendente.component*** e quindi è visibile da ogni altra maschera che si decide di andare a visualizzare all'atto della selezione di una voce dal menu laterale.

Per gestire il salvataggio è stato definito nel servizio **EmployeeState**, già utilizzato nella descrizione nella maschera contente gli storici, un Observable che viene usato per notificare le maschere, nelle quali è possibile andare ad effettuare un'operazione di modifica, che si è attivato il meccanismo di salvataggio dei dati. La *Figura 27* mostra il codice e il metodo per fare ciò.

```
// Observable string source
private saveEmployeeSource = new Subject<boolean>();

// Observable string stream
saveEmployee$ = this.saveEmployeeSource.asObservable();

public setEmployeeSave(message: boolean) {
  this.saveEmployeeSource.next(message);
}
```

Figura 27: Definizione Observable per attivare il salvataggio dei dati

Si osserva che per la notifica ai componenti interessati nel processo di salvataggio si invia una variabile booleana.

A questo punto, nel componente ***dipendente.component*** è stata definita la funzione ***save()*** che si attiva al click sul bottone "Salva" presente nel template del medesimo componente e che non fa altro che andare a settare una variabile booleana a true e ad attivare il metodo ***setEmployeeSave()*** del servizio mostrato nella

Figura 27, passandogli come parametro la variabile appena valorizzata.
Siccome gli unici componenti nei quali è possibile effettuare le modifiche sono quello contenente i dati anagrafici/aziendali e quello riferito agli storici, è necessario agganciare questi componenti all'Observable definito mediante il metodo ***save()*** del componente *dipendente.component*.
Questo avviene effettuando una sottoscrizione, mediante il metodo ***subscribe()*** nel costruttore dei due componenti che devono inviare i dati per poi andarli a salvare.
Si può osservare in *Figura 28* la sottoscrizione del componente ***storici.component*** allo stream *saveEmployee$*.

```
this.employeestate.saveEmployee$.subscribe(savingEmployeeData => {
  if (savingEmployeeData) {
    this.employeestate.sendEmployeeStoriciToParent(this.employeeStorico);
  }
})
```

Figura 28: Sottoscrizione del componente *storici.component*

Si nota che viene invocato il metodo ***sendEmployeeStoriciToParent()*** il cui codice viene riportato in *Figura 29*.

```
private EmployeeStoriciToParentSource=new Subject<EmployeeViewModelDTO>();

EmployeeStoriciToParent$=this.EmployeeStoriciToParentSource.asObservable();

public sendEmployeeStoriciToParent(employeeStorici: EmployeeViewModelDTO) {
  this.EmployeeStoriciToParentSource.next(employeeStorici);
  this.EmployeeStoriciToParentSource.complete();
}
```

Figura 29: Codice per inviare i dati da *storici.component* a *dipendente.component*

Il metodo ***complete()***, come già definito nel capitolo contente gli strumenti utilizzati, viene eseguito per segnalare che l'inoltro di dati è terminato.
Invece, in *Figura 30*, si nota la sottoscrizione all'Observable da parte del componente contenente i dati anagrafici. Infatti, dato che la porzione di template contenente i dati aziendali presenta campi di sola lettura, non può subire modifiche e quindi è inutile inviare i dati per il salvataggio.

```
this.employeestate.saveEmployee$.subscribe(savingEmployeeData => {
  if(savingEmployeeData){
    this.employeestate.sendAnagraficaToParent(this.employeeAnagrafica);
  }
});
}
```

Figura 30: Sottoscrizione del componente *datiAnagrafici.component*

In questo caso si utilizza la funzione ***sendAnagraficaToParent()*** che, già dal nome, fa capire che svolge le stesse funzionalità di ***sendEmployeeStoricoToParent()***, della quale per completezza viene riportato il codice in *Figura 31*.

```
// Observable string source
private AnagraficaToParentSource = new Subject<EmployeeViewModelDTO>();

// Observable string stream
AnagraficaToParent$ = this.AnagraficaToParentSource.asObservable();

public sendAnagraficaToParent(employeeAnagrafica: EmployeeViewModelDTO) {
  this.AnagraficaToParentSource.next(employeeAnagrafica);
  this.AnagraficaToParentSource.complete();
}
```

Figura 31: Codice per inviare i dati da *datiAnagrafici.component* a *dipendente.component*

Quindi mediante queste due porzioni di codice all'atto della creazione dei due componenti essi si sottoscrivono allo stream di dati definito nel componente *dipendente.component*. Tuttavia solo all'atto del click sul pulsante "Salva" viene passata a questo stream una variabile booleana il cui valore notifica ai due componenti che sono agganciati a questo stream che è iniziato il procedimento di salvataggio e che quindi devono inviare i dati da salvare. In particolare, il componente ***datiAnagrafici.component*** invia in questo stream aolo i dati relativi all'anagrafica, mentre il componente ***storici.component*** invia tutti quelli del dipendente, aggiornati a seguito di eventuali modifiche o inserimento di righe.

Siccome si ha una situazione nella quale ci sono più Observable, in particolare uno viene usato per notificare i componenti del salvataggio e due vengono utilizzati per inviare i dati, per gestire questa situazione nel costruttore del componente ***dipendente.component*** si utilizza l'operatore **forkJoin**[11].

Si tratta di un operatore della libreria *RxJS* che viene utilizzato quando si ha un gruppo di osservabili, ma si è interessati solo al valore finale emesso da ciascuno. Un caso d'uso comune per questo operatore è quello in cui si desiderano inviare eventi a più componenti, attivando l'invio solo quando è stata ricevuta una risposta da

tutti. Questa situazione è associata ad un meccanismo di salvataggio nel quale i dati provengono da componenti diversi fra di loro, in quanto si deve procedere al salvataggio solamente quando tutti sono stati ricevuti, altrimenti si rischierebbe di perdere alcune informazioni all'atto del nuovo caricamento della pagina.
Il codice relativo all'operatore presente nel costruttore del componente ***dipendente.component*** è mostrato in *Figura 32*.

```
forkJoin(this.employeestate.AnagraficaToParent$, this.employeestate.EmployeeStoriciToParent$).subscribe(([Anagrafica, Storico])=> {
  let employeeWriteModel: EmployeeViewModelDTO={
    id:Anagrafica.id,
    employeeCode:Anagrafica.employeeCode,
    employeeName:Anagrafica.employeeName,
    employeeSurname:Anagrafica.employeeSurname,
    authorityEmployeeCode:Anagrafica.authorityEmployeeCode,
    gender:Anagrafica.gender,
    birthDate:Anagrafica.birthDate,
    idBirthMunicipalityAdministrativeArea:Anagrafica.idBirthMunicipalityAdministrativeArea,
    employeeCF:Anagrafica.employeeCF,
    phoneNumber:Anagrafica.phoneNumber,
    mobilePhoneNumber1:Anagrafica.mobilePhoneNumber1,
    mobilePhoneNumber2:Anagrafica.mobilePhoneNumber2,
    operatorEmail:Anagrafica.operatorEmail,
    personalEmail:Anagrafica.personalEmail,
    employeePropertyInEmployee: Storico.employeePropertyInEmployee
  }
  this.employeeservice.saveEmployee(employeeWriteModel)
  window.location.reload();
});
```

Figura 32: Operatore forkJoin

In questo caso, l'operatore ***forkJoin*** prende come parametri due osservabili riferiti allo stream di dati che vengono inviati dai componenti ***datiAnagrafici.component*** e ***storici.component*** e, la sottoscrizione con *subscribe* viene attivata solo nel momento in cui tutti i dati emessi sono disponibili. Questo è il motivo per cui, come mostrato in *Figura 29* e in *Figura 31* si utilizza il metodo *complete()*. Infatti, fintanto che un osservabile non completa l'invio dei dati, l'operatore *forkJoin* non emetterà alcun valore. L'operatore *forkJoin*, a sua volta, è un Observable a cui è possibile effettuare una sottoscrizione andando ad elaborare i dati che sono stati ricevuti. All'interno del metodo *subscribe()* viene definito il vettore *[Anagrafica, Storico]* dove i due valori rappresentano un alias per i due osservabili definiti come parametro di *forkJoin*. Come prima parte del metodo si crea un oggetto di tipo *EmployeeViewModelDTO* che rappresenta la struttura dati che contiene tutte le informazioni del dipendente. Successivamente si valorizzano i campi relativi all'anagrafica con ciò che viene inviato dallo stream *AnagraficaToParent$* e quelli relativi ai dati aziendali con ciò che viene inviato dallo stream *EmployeeStoriciToParent$*.
Infine viene attivato il metodo ***saveEmployee()*** e successivamente si va a ricari-

care la pagina in modo da poter visualizzare i dati che sono stati aggiornati.
Questa funzione si trova all'interno del servizio *EmployeeService* che è lo stesso che contiene il metodo *getEmployee()* che viene invocato al caricamento della maschera del dipendente.
Come già suggerisce il nome, la funzione ***saveEmployee()*** procede andando a salvare i dati del dipendente. Il codice viene riportato in *Figura 33*.

```
public saveEmployee(employeeWriteModel: EmployeeViewModelDTO): Observable<EmployeeViewModelDTO> {
  const headers = new HttpHeaders().set('Content-Type', 'application/json');
  return this.http.post<EmployeeViewModelDTO>(`${this.appConfigService.urlBackEnd}/employee`,
    employeeWriteModel,
    { headers: headers }).pipe(
      catchError((error) => {
        console.log('POST: Error saving the provided employeeservice', employeeWriteModel, error);
        return throwError(error);
      }));
  }
}
```

Figura 33: Funzione salvataggio dati

La funzione prende come parametro un oggetto che contiene tutti i dati del dipendente e procede definendo una chiamata di tipo POST alla funzione lato back-end che aggiorna i dati. Si osserva che l'URL non contiene l'identificatore del dipendente e questo perchè esso può essere recuperato dalla funzione lato server grazie alle proprietà contenute nell'oggetto passato come parametro al metodo stesso.

5.6 Maschera contenente i KPI

In quest'ultima sezione si descriverà la maschera contenente le voci relative ai KPI e alle loro statistiche, con la possibilità di visualizzarne il loro andamento attraverso un grafico. Si ricorda che KPI è l'acronimo di *Key Performance Indicators*, che rappresentano gli indici di un processo aziendale, cioè misurano il volume del lavoro di un processo.

Anche in questo caso, prima di procedere andando a descrivere puntualmente le funzionalità della maschera in questione, si decide di mostrare le varie interfacce a seconda delle possibili interazioni che l'utente può avere con la maschera.

All'atto della selezione della voce "KPI" dal menu laterale viene caricata la sezione desiderata, che appare come mostrato in *Figura 34*.

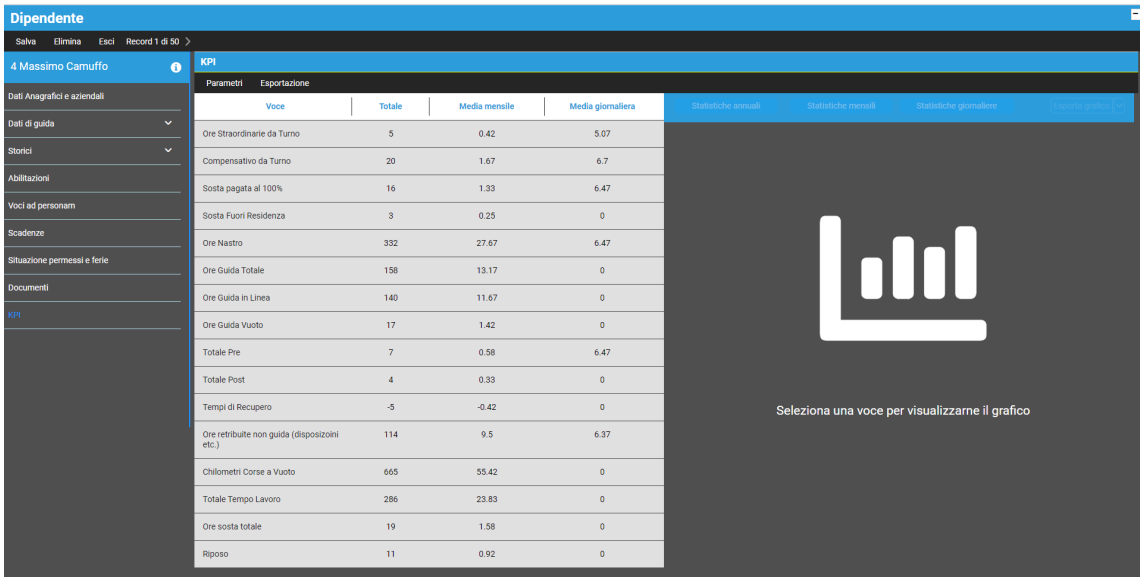

Figura 34: Maschera KPI senza aver selezionato alcuna voce

Si osserva che la maschera è suddivisa in due sezioni. Quella a sinistra contiene una tabella con una serie di voci che rappresentano i KPI e di ognuno si hanno i valori circa il totale, la media mensile e quella giornaliera. In particolare, queste due ultime voci nel momento in cui l'applicativo Motus verrà terminato dovranno fare riferimento al mese immediatamente precedente a quello corrente, e ai relativi giorni.

Nella sezione di destra si trova inizialmente un messaggio che invita l'utente a selezionare le voci in modo da poterne visualizzare il relativo grafico e ovviamente

fintanto che questa operazione non avviene, i bottoni in alto in cui è possibile cambiare il grafico che si sta visualizzando sono disabilitati.
Quello che accade nel momento in cui si seleziona un voce da quelle presenti viene mostrato in *Figura 35*.

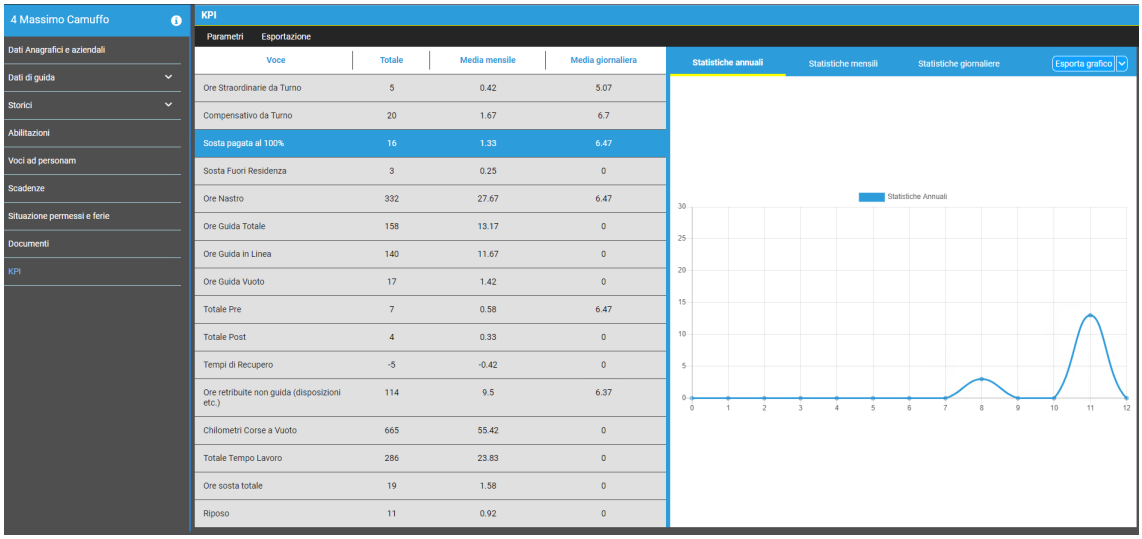

Figura 35: Maschera KPI nel momento in cui si seleziona una voce

Quindi nel momento in cui si seleziona una qualsiasi voce dalla griglia di quelle presenti la relativa riga viene evidenziata, affinchè l'operatore riesca a capire a quale voce fa riferimento il grafico che sta visualizzando, e di questa voce selezionata viene mostrato inizialmente il grafico relativo alle statistiche annuali. In questo caso, il diagramma presenta in ascissa una serie di numeri che vanno da 0 a 12 e che rappresentano i mesi dell'anno corrente, dove il numero 0 fa riferimento a Dicembre dell'anno precedente. Inoltre, si osserva che in questo caso i bottoni presenti sopra al grafico sono abilitati, in modo che è possibile visualizzare, sempre relativamente alla stessa voce, anche le statistiche mensili e giornaliere. In riferimento a quest'ultimo caso, nel progetto non è stata implementata la visualizzazione delle statistiche giornaliere nelle quali il grafico però troverebbe in ascissa le ore della giornata. Nel momento in cui si clicca il bottone relativo alle statistiche mensili, viene mostrato un nuovo grafico, come si osserva in *Figura 36*.

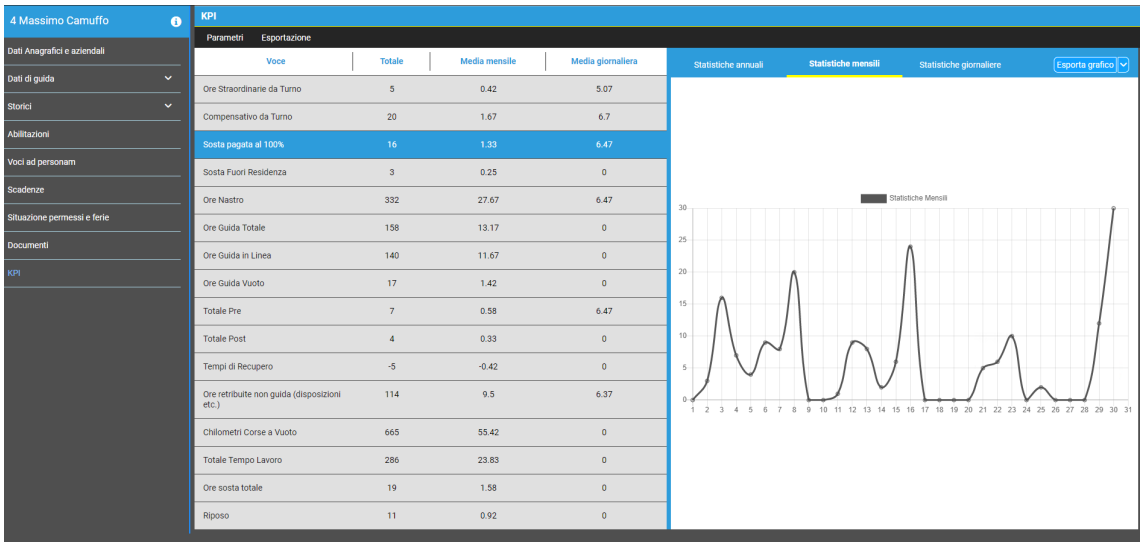

Figura 36: Maschera KPI nel momento in cui si selezionano le statiche mensili

In questo caso, il grafico che viene mostrato presenta in ascissa i giorni del mese e ad ognuno di essi compare un relativo valore.

A questo punto si può procedere andando ad analizzare le caratteristiche e le funzionalità della maschera.
A livello di template i grafici sono stati realizzati mediante il componente **p-chart** di *PrimeNG* il quale presenta un attributo *data* nel quale va definito l'oggetto che contiene tutte le caratteristiche dei dati che dovranno essere rappresentati e del grafico stesso. Ad esempio, quando si vogliono visualizzare i dati relativi alle statistiche annuali si valorizza l'attributo *data* come segue:

```
[data]=dataYear
```

dove *dataYear* viene mostrato in *Figura 37*.

```
this.dataYear = {
  labels: month,
  datasets: [
      {
          label:'Statistiche Annuali',
          data: this.dutyItemAmount,
          fill: false,
          borderColor: '#2d9cdb'
      }

  ]
}
```

Figura 37: Oggetto utilizzato per mostrare i dati graficamente

Per completezza si procede andando a descrivere i vari campi:

- **labels**: contiene l'array che viene utilizzato per popolare l'asse delle ascisse;
- **datasets**: contiene l'oggetto (o gli oggetti) usati per popolare il grafico. In particolar modo:
  - **label**: definisce l'etichetta associata alla linea usata per visualizzare l'andamento del grafico;
  - **data**: definisce l'array contenente i dati da visualizzare;
  - **fill**: permette di riempire l'area tra il grafico e l'asse delle ascisse;
  - **borderColor**: definisce il colore della linea associata al grafico.

All'atto del caricamento della maschera si valorizzano gli array *employeeEntriesMonthly* e *employeeEntriesDaily*, associati entrambi al decoratore @Input(), rispettivamente con i dati relativi ai diversi mesi dell'anno e ai differenti giorni dei mesi.
A questo punto si procede andando a vedere in che modo si popolano le varie voci della tabella posta a fianco del grafico.
L'idea per andare a valorizzare i vari campi è stata di creare un array di elementi, dove ogni elemento presenta quattro campi relativi alla voce, al totale, alla media mensile e a quella giornaliera e poi a livello di template scandire ogni elemento di questa struttura dati, mediante la direttiva **ngFor*, per poi estrarre i valori di interesse.
La creazione di questo array è definita nel metodo **ngOnInit()** e viene mostrata in *Figura 38*.

```
this.idDutyItemRecords = _.uniqBy(this.employeeEntriesMonthly,"idDutyItem");
this.idDutyItemRecords.forEach(element => {
  element.total=Math.round(this.calcoloTotale(element.idDutyItem)*100)/100;
  element.avgMonth=Math.round(this.calcoloTotale(element.idDutyItem)/12*100)/100;
  element.avgDay=this.avgMonthCount(element.idDutyItem);

});
```

Figura 38: Creazione dell'array di oggetti per valorizzare i campi della griglia

La variabile *idDutyItemRecords* rappresenta l'array di oggetti usati per riempire i vari campi della griglia.
La prima istruzione utilizza la funzione *Lodash* **_.uniqBy** la quale prende come parametro l'array di oggetti *employeeEntriesMonthly*, che contiene i dati relativi ai

diversi mesi dell'anno per tutti i KPI, e come secondo parametro il codice relativo ad ogni singolo oggetto dell'array. Quello che restituisce tale funzione è un nuovo array di oggetti, dove però tra tutti gli elementi che presentano un *idDutyItem* identico ne viene preso solo uno e quindi tale struttura dati non presenterà duplicati rispetto tale proprietà.

Successivamente si applica all'array il metodo JavaScript ***forEach()*** che permette di scandire ogni elemento dell'array assegnandogli la variabile di nome *element* e andando ad estendere le proprietà dell'elemento in cui questione definendone tre nuove che si riferiscono alle voci "Totale", "Media mensile" e "Media giornaliera". Ognuna di esse viene definita mediante una funzione ad hoc, in particolare per il totale si applica la funzione ***calcoloTotale()***, che prende come parametro il codice associato al KPI e scorre di volta in volta tutto l'array *employeeEntriesMonthly* sommando tra loro i valori che presentano un identificatore identico a quello passato come parametro.

Per il calcolo della media mensile è sufficiente dividere il risultato per 12, mentre per il calcolo della media giornalieria si fa riferimento alla funzione ***avgMonthCount()***. Essa prende come parametro il codice identificativo del KPI e, scorrendo l'intero array *employeeEntriesDaily*, va a prelevare di ogni elemento con identificatore uguale a quello passato come parametro il mese associato al dato incrementando ogni volta il valoredella variabile *total* solo se il mese corrisponde a Novembre. A questo punto al termine del ciclo in questa variabile si ha il totale dei valori associati al KPI nel mese indicato e come ultima operazione si calcola la media giornaliera suddividendo il risultato per 30, che rappresentano i giorni che ha il mese di Novembre.

In questo modo sono stati analizzati tutti i meccanismi che si attivano per andare a generare e riempire la griglia nella parte a sinistra della maschera. Perciò si può andare a descrivere ciò che succede nel momento in cui si seleziona una voce per visualizzare graficamente l'andamento dei valori.

Come già definito precedentemente, al click su una delle voci della tabella compare il grafico relativo alle statistiche annuali. Per fare questo viene attivato il metodo ***showEmployeeChartAnnuale()*** che prende come parametro l'identificatore del KPI in questione e procede andando a valorizzare la variabile *dutyItemAmount* che, come già mostrato in *Figura 40*, viene utilizzata dal grafico per rappresentare i

valori. Siccome tale variabile viene utilizzata sia per la visualizzazione delle statistiche annuali che mensili, nella funzione ***showEmployeeChartAnnuale()***, prima si azzerano tutti i valori e poi si valorizzano nuovamente in modo opportuno, come mostrato dal codice in *Figura 39*.

```
for(let i=0; i<this.employeeEntriesMonthly.length; i++) {
  if(this.employeeEntriesMonthly[i].idDutyItem==idDutyItem) {
  let dayStartDate = new Date(this.employeeEntriesMonthly[i].dayStart);
  this.dutyItemAmount[dayStartDate.getMonth()+1]+=this.employeeEntriesMonthly[i].dutyItemAmount;
  }
}
```

Figura 39: Codice che permette di visualizzare il grafico relativo alle statistiche annuali

Quindi, ciò che viene fatto è di selezionare ogni oggetto all'interno dell'array *employeeEntriesMonthly* e poi, per quelli che presentano un codice identificativo identico a quello che viene passato come parametro alla funzione, definire un oggetto di tipo *Date* in modo da poter poi estrarre il mese di riferimento così da utilizzarlo come indice per accedere al corrispondente valore nell'array *dutyItemAmount* e incrementarlo.
In questo modo, dopo aver selezionato una voce, si ha un array dove gli indici rappresentano i mesi e i valori corrispondenti sono quelli che poi vengono mostrati graficamente.

Nel momento in cui, dalla voce già selezionata, si preme il bottone relativo alle statistiche mensili, si attiva il metodo ***showEmployeeChartMensile()*** il quale prende sempre come parametro il codice identificativo relativo al KPI di interesse. Come già detto prima, per rappresentare i dati graficamente si utilizza sempre il vettore *dutyItemAmount*, azzerandolo preliminarmente per poi valorizzarlo in modo opportuno, come viene mostrato in *Figura 40*.

```
for(let i=0; i<this.employeeEntriesDaily.length; i++) {
  if(this.employeeEntriesDaily[i].idDutyItem==idDutyItem) {
    let dayDate = new Date(this.employeeEntriesDaily[i].day);
    if(dayDate.getMonth()==10) {
      this.dutyItemAmount[dayDate.getDate()]+=this.employeeEntriesDaily[i].dutyItemAmount;
    }
  }
}
```

Figura 40: Codice che permette di visualizzare il grafico relativo alle statistiche mensili

Rispetto al caso precedente, si scansionano uno ad uno gli oggetti dell'array *employeeEntriesDaily* e per quelli che presentano un codice identificativo uguale a quello che viene passato come parametro si istanzia un oggetto della classe *Date* per poi estrarne il giorno, mediante la funzione ***getDate()***, e utilizzarlo come indice per accedere al valore dell'array *dutyItemAmount* ed incrementarlo.
Quindi analogamente a prima, in tale array troveremo che gli indici rappresentano i giorni del mese di riferimento, e i valori verranno utilizzati per creare il grafico.

Capitolo 6

Conclusione e Sviluppi futuri

Motus rappresenta un applicativo ancora in via di sviluppo all'interno dell'azienda **Pluservice S.r.l.** e data la numerosità di funzionalità che presenta al suo interno il progetto è stato focalizzato solo su alcune di esse, in particolare sulla implementazione, nonchè gestione, dei meccanismi associati alla maschera del dipendente.
Come si è potuto notare dai capitoli precedenti, in particolare quello riferito all'applicazione sviluppata (*Capitolo 5*), si è trattata la gestione della parte front-end dell'applicazione e quindi, dove necessario, si sono utilizzati i servizi messi a disposizione da altri sviluppatori dell'azienda.
Riguardo l'utilizzo delle tecnologie, e in particolar modo del framework Angular, erano già state prefissato e questo a causa del fatto che per quanto Motus rappresenta un'applicazione ancora in via di sviluppo il seguente progetto è stato inserito all'interno di un ambiente già esistente. Tuttavia, l'utilizzo di questo framework ha permesso di poter approfondire alcune delle tecnologie che nel corso di laurea non sono state affrontate, che però sono state integrate facilmente con le conoscenze acquisite durante il corso di Tecnologie Web.
La scelta di affrontare un tirocinio associato alla tesi ha permesso di poter osservare da vicino il mondo lavorativo e soprattutto lo sviluppo di applicazioni Web complesse, che ad oggi sono necessarie per la gran parte della soluzione dei problemi moderni.
Inizialmente, circa gli obiettivi di progetto, era stato accennato di sviluppare anche la maschera relativa alla sezione dei "Dati di guida", tuttavia, a causa del numero di ore associate al tirocinio, questo non è stato possibile. Fatto sta che, i restanti re-

quisiti progettuali sono stati soddisfatti pienamente, in particolar modo, riguardo le funzionalità che si sono dovute implementare, grazie anche alle conoscenze acquisite negli altri corsi, si è potuta sviluppare autonomamente una porzione di applicazione partendo da un'attenta analisi dei requisiti, seguita dalla progettazione, fino allo sviluppo vero e proprio con i relativi test finali. Invece, circa lo sviluppo dell'interfaccia utente si sono seguiti dei mockup sviluppati da un designer e quindi su questo aspetto si è cercato di realizzare un template il più coerente possibile con quanto prestabilito, sfruttando le note tecnologie HTML e CSS.

Come già affermato precedentemente, la soluzione proposta si presenta all'interno di un'applicativo non ancora in commercio per il quale possono essere ipotizzate ulteriori funzionalità, come ad esempio fornire all'operatore la possibilità di accedere ad una sezione direttamente all'interno della maschera del dipendente in cui poter visualizzare in tempo reale quali sono le tratte che l'autista sta svolgendo senza accedere alla sezione apposita, con possibilità di eventuali modifiche. Oppure, riguardo la sezione contenente i KPI, visto che sono stati utilizzati dei grafici per visualizzare delle statistiche, sarebbe interessante dare la possibilità all'operatore di poter confrontare le statistiche fra il dipendente associato alla maschera e uno o più dipendenti fra quelli disponibili, in modo da poter studiare i diversi indici anche sulla base di un confronto fra differenti dipendenti.

Bibliografia

[1] https://vitolavecchia.altervista.org/sistemi-informativi-integrati-erp-enterprise-resource-planning - consultato a Maggio 2020

[2] Coppola Pierluigi, Silvestri Fulvio - "Sistemi avanzati di gestione del trasporto pubblico" - 2018

[3] https://angular.io/guide/architecture-components - consultato a Maggio 2020

[4] https://angular.io/guide/lifecycle-hooks - consultato a Maggio 2020

[5] https://www.html.it/pag/60329/direttive-in-angular-2/ - consultato a Maggio 2020

[6] https://angular.io/guide/architecture-services - consultato a Maggio 2020

[7] https://angular.io/guide/observables - consultato a Maggio 2020

[8] https://www.primefaces.org/primeng/showcase/#/ - consultato a Maggio 2020

[9] http://www.datrevo.com/lodash-un-utile-libreria-javascript/ - consultato a Maggio 2020

[10] https://lodash.com/docs/4.17.15 - consultato a Giugno 2020

[11] https://www.learnrxjs.io/learn-rxjs/operators/combination/forkjoin - consultato a Giugno 2020