UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

UNIVERSITÀ DEGLI STUDI DI MODENA

E REGGIO EMILIA

Dipartimento di Ingegneria "Enzo Ferrari"

Corso di Laurea Magistrale in Ingegneria Informatica (LM-32)

Analisi e sviluppo di un'applicazione cloud per la gestione e l'efficientamento della gestione licenze del prodotto GPOne e dei flussi interni dell'azienda OSL S.r.l.

Relatore:
Prof. Riccardo Lancellotti

Candidato:
Arturo Bianchi
Matricola 181000

ANNO ACCADEMICO 2024/2025

Indice

Elenco delle Figure

Elenco dei Codici

Introduzione

Il presente lavoro di tesi analizza tutte le fasi di progettazione e di sviluppo per un applicativo gestionale interno realizzato presso l'azienda OSL S.r.l., il cui scopo è l'ottimizzazione e la modernizzazione del processo di gestione licenze software del prodotto GPOne . Nel contesto tecnologico attuale, per un'azienda è fondamentale poter sfruttare per i propri prodotti i modelli di vendita più adatti per il mercato. Questo per l'azienda non è stato sempre possibile, complice un retaggio un po' arretrato di gestione e di prodotti aziendali preesistenti.

L'esigenza del progetto nasce dall'analisi dei processi preesistenti all'interno dell'azienda. Tali processi sono stati concepiti inizialmente per permettere un time-to-market di GPOne il più basso possibile e, seppure funzionali allo scopo, questo ha portato all'utilizzo di tecnologie in parte obsolete e inadeguate a una scalabilità e un'evoluzione sia dei processi che del prodotto stesso, causando più volte rallentamenti e problemi nelle installazioni cliente e nel fornire assistenza di primo e secondo livello.

L'obiettivo dell'elaborato è quindi illustrare in primo piano la realizzazione di una soluzione software adeguata a facilitare il rilascio e il monitoraggio delle licenze, andando dunque a supportare meglio i processi sia in fase di vendita, che di installazione ed assistenza dei vari reparti commerciali, manageriali e tecnici. Inoltre verranno mostrati i primi passi che l'azienda ha mosso nell'ottica dello sviluppo in cloud, in particolare sulle infrastrutture di Microsoft Azure, al fine di poter dare una scelta più moderna sia per lo sviluppo che per l'efficientamento dei processi.

Nei seguenti capitoli verranno dunque analizzati:

- L'analisi dei requisiti, lo studio del dominio applicativo ed il contesto normativo preso in esame nel periodo di analisi e sviluppo;
- Le scelte architetturali, infrastrutturali e di progettazione dei flussi di gestione.
- Le fasi di sviluppo, testing, implementazione e integrazione delle varie funzionalità nel già esistente contesto software aziendale.
- I risultati ottenuti in termini di efficienza dei processi aziendali, con analisi dell'esito delle prime installazioni effettuate.

1. Analisi iniziali e contesti

1.1 Il contesto aziendale di OSL S.r.l.

OSL S.r.l. è una software house italiana che si dedica allo sviluppo di soluzioni digitali per la smart factory, dedicate alla gestione e al monitoraggio della produzione, alla raccolta dei dati e ai sistemi CAD/CAM per le aziende meccaniche.

L'azienda nasce nel 1991 ed oggi con oltre 30 anni di esperienza conta oltre 100 dipendenti e più di 3000 clienti su tutto il territorio italiano.

Nel 2019 è stata inglobata dal Gruppo Overmach S.p.A., leader nel settore delle macchine utensili e dell'utensileria con sede a Parma.

OSL vende una serie di prodotti software, oltre a vari servizi di consulenza diretta, sia tecnica che di gestione dei processi aziendali. I prodotti però che più trainano l'azienda sono due:

- la suite GP9Over, storico software gestionale dell'azienda che ha un ciclo di vita di oltre 30 anni e permette la gestione di vari aspetti della tipica azienda metalmeccanica, dalla gestione magazzino a tutta la parte MES-ERP;
- la suite GPOne, il nuovo software web-based venduto al pubblico dal 2024 con l'obiettivo di andare a sostituire nel tempo GP9Over, ancora nelle prime iterazioni di sviluppo, al momento viene principalmente venduto come MES e per la gestione programmi delle macchine.

Ad eccezione di alcuni prodotti lanciati negli ultimi mesi, l'azienda ha sempre sviluppato software on-premises e la vendita dei vari prodotti e dei relativi moduli è sempre stata effettuata tramite la creazione di una licenza perpetua.

La maggior parte della clientela di OSL fa parte delle PMI metalmeccaniche, quindi tutte quelle aziende medio-piccole che rappresentano la maggioranza delle aziende italiane, tuttavia tra le fila dei clienti sono presenti anche alcune grandi aziende.

La struttura aziendale è verticale; vi sono 4 macro-reparti con un responsabile d'area per ognuno di essi e a loro volta sono suddivisi in reparti con ognuno un responsabile di reparto.

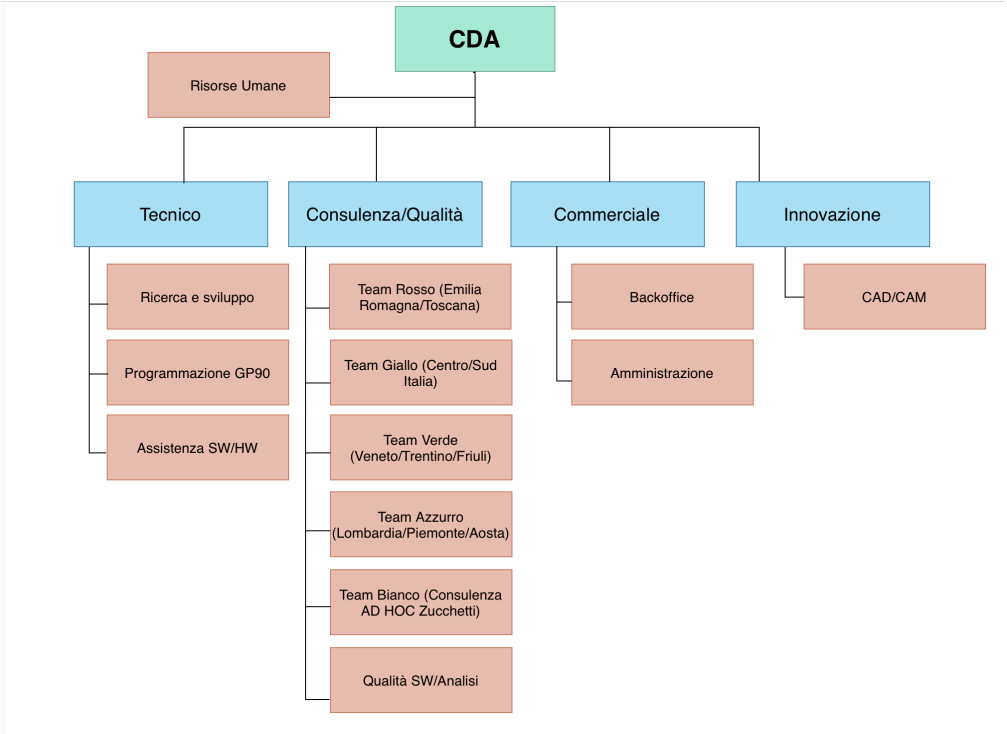

Figura 1.1: Struttura aziendale di OSL

Per i punti che verranno affrontati in seguito nell'elaborato è necessario descrivere un minimo alcuni reparti d'interesse. Verranno ignorati il reparto Risorse Umane, che è autoesplicativo, e il reparto Innovazione, dato che non sono stati coinvolti tra gli stakeholders iniziali e nello sviluppo.

Il reparto Consulenza è il reparto più folto dell'azienda, vanta oltre 40 dipendenti tra team leaders e consulenti, e ha come compito principale l'installazione dei prodotti software on-premises dal cliente e l'analisi iniziale delle richieste cliente e delle user stories, nonchè fornire assistenza di primo e secondo livello e talvolta anche lo sviluppo di alcune personalizzazioni semplici. Il reparto si divide in 4 squadre che gestiscono zone diverse del territorio italiano per i prodotti sviluppati in casa da OSL; fa eccezione la quinta, il team Bianco, che è specializzata nel supporto di AD HOC di Zucchetti, di cui OSL è rivenditore.

Come si può vedere dall'immagine 1.1, sotto l'area della consulenza è presente anche il reparto Qualità software e analisi. Questo si occupa principalmente di fornire le analisi funzionali al reparto di Ricerca e Sviluppo e di effettuare test sugli ambienti di prova e sulle versioni pre rilascio dei software sviluppati.

Il reparto Commerciale contiene circa 15 dipendenti, tra agenti commerciali, impiegati di

backoffice e amministrazione. I compiti principali riguardano le offerte di vendita ai clienti, la stesura dei contratti di acquisto e l'inserimento degli ordini cliente. Gli agenti, in sinergia con il reparto consulenza che fa una prima analisi e successivamente con il reparto Ricerca e Sviluppo o Programmazione, propone una serie di prodotti e relativi moduli nell'offerta. Se l'offerta viene accettata, la parte burocratica di gestione dell'ordine interno viene poi gestita dal backoffice.

Il reparto tecnico presenta circa 30 dipendenti tra assistenza e sviluppo.

L'Assistenza si occupa di fornire supporto di primo livello al cliente finale soprattutto nelle fasi post installazione software e formazione cliente, talvolta possono anche sostituire i consulenti tecnici nella fase di installazione, specialmente per i prodotti facenti parte della suite di GPOne.

La Programmazione GP90 si occupa di assistenza di secondo livello relativa alla suite di GP9Over e degli sviluppi; essendo un software a fine ciclo di vita e siccome i commerciali cercano di venderlo il meno possibile, la maggior parte degli sviluppi sono relativi a integrazioni specifiche con terze parti.

La Ricerca e Sviluppo è il reparto che si occupa della suite di GPOne e dei software più recenti dell'azienda ed è il reparto in cui sono inserito; la maggior parte del carico di lavoro attuale riguarda lo sviluppo di nuove feature relative al mondo ERP e al mondo della gestione dei magazzini, tuttavia con l'aumento delle vendite del prodotto il supporto alle prime installazioni inizia a portare un carico non più trascurabile.

L'ultimo reparto che si descriverà non appare nel grafico di figura 1.1, poichè non fa parte di OSL, tuttavia è necessario introdurlo perchè farà parte di alcuni aspetti dello sviluppo che verranno introdotti nel capitolo 2. Il CED di Overmach ha in gestione tutta la parte I.T. sia di Overmach che di OSL. Dunque tra le mansioni, ci sono tutte le questioni relative alla sicurezza, quindi gestione dei firewall e dei perimetri aziendali, la gestione e la predisposizione dei dispositivi aziendali quali pc e cellulari, la gestione di software e licenze di terze parti usati per la gestione aziendale e per gli sviluppi.

1.2 Motivazioni della nuova gestione licenze

L'esigenza dello sviluppo di un nuovo software per gestire le licenze nasce da più motivazioni.

Abbiamo accennato al fatto che l'azienda abbia sempre ragionato con l'idea di vendere i software sviluppati solamente tramite l'acquisto di una licenza perpetua, con l'aggiunta di moduli anch'essi senza data di scadenza. L'approccio ha chiaramente dei benefici: lato cliente l'esborso economico viene effettuato solo una volta all'inizio, e poi a seguire solo per eventuali aggiunte di nuovi moduli; lato OSL, almeno sulla vendita del software, non c'è il rischio di clienti insolventi, dato che il software viene installato solamente una volta effettuata la vendita del prodotto.

Con l'uscita della suite di GPOne però hanno cominciato a presentarsi anche una serie di problematiche che non sono più ignorabili. Dato che GPOne è stato pensato per essere in un futuro vendibile anche come SaaS, dev'essere possibile fornire una licenza gestibile da remoto, in automatico, che possa avere delle date di scadenza sui singoli moduli. La gestione da remoto consentirebbe di snellire enormemente il tempo di consegna di un nuovo modulo attivato e, un domani, l'installazione dell'intero applicativo. Inoltre con la vendita dei servizi in abbonamento, sarebbe possibile anche bloccare in modo tempestivo l'utilizzo del software qualora ci fossero eventuali problemi, come clienti insolventi o disdetta da parte di quest'ultimo perchè passato ad un altro prodotto,

La gestione da remoto permetterebbe anche un tracciamento dei clienti attivi migliore. Un problema che si è presentato in azienda riguardo le installazioni di GP9Over è stato il censimento di quali clienti utilizzano ancora il gestionale, e come risultato dell'indagine che è stata condotta è risultato che un terzo dei clienti che sono censiti nel database sono ormai aziende chiuse o che sono passati ad altri gestionali. Internamente questi vengono definiti clienti "zombie".

Chiaramente un controllo da remoto non è sempre possibile: ci sono casi in cui il cliente, qualora sia soggetto a standard di sicurezza molto stringenti, non rende possibile il raggiungimento di un SaaS esterno alla rete aziendale, anche nel caso in cui il provider dell'host del software sia in linea con gli standard di sicurezza più richiesti, come possono essere AWS o Azure; anche qualora il cliente abbia l'applicativo su un suo server on-premises, potrebbe non accordare l'utilizzo di una porta per consentire attraverso il firewall la comunicazione tra il servizio remoto e il server interno.

Un'altra problematica che si è presentata sulle installazioni di GPOne riguarda il blocco della licenza qualora cambiasse qualcosa nel server cliente. La licenza che viene data ai consulenti per effettuare l'installazione viene utilizzata solo in fase d'installazione. In fase di setup, tale file di licenza viene sostituito da un file cifrato tramite una chiave a 16 bit composta da una combinazione

dei seriali di alcuni componenti del server. Tali componenti cambiano a seconda del sistema operativo, ad esempio su Windows si è scelto di usare una combinazione dei seriali di processore, scheda madre e disco rigido.

Sebbene sia una soluzione abbastanza robusta nel caso di tentativo di passaggio licenza a un altro server in modo illecito, ci sono stati casi in cui il cliente ha dovuto fare dei cambi server e si è ritrovato bloccato. Uno dei casi più comuni che si è presentato è stato la modifica di una macchina virtuale che il cliente ha fornito per l'installazione di GPOne; il cliente in questo caso ha avuto necessità di migrare le macchine virtuali, oppure ha semplicemente dovuto allocare o deallocare risorse alla macchina virtuale, magari fornendo più core al processore o più disco, ma questo ha provocato un cambio nel seriale che si andava a leggere. Il risultato è stato un blocco di tutti i microservizi di GPOne e dunque di un blocco del monitoraggio della produzione, con relativo ticket di assistenza che è stato aperto.

La risoluzione in questo caso consisteva nel contattare direttamente uno sviluppatore della Ricerca e Sviluppo che si collegava da remoto al cliente e ricreava il nuovo file cifrato della licenza, causando rallentamenti agli sviluppi che venivano portati avanti in quel momento. Questo risoluzione non poteva passare dal consulente del cliente, perchè per disposizione interna si è scelto di non dare la possibilità ai consulenti di usare il tool, per evitare che si diffondesse troppo; nemmeno l'assistenza poteva fornire questo tipo di supporto, sebbene non fosse complicato, ma essendo già un reparto sotto un importante carico di lavoro dovuto ai software preesistenti, per questi primi clienti di GPOne si è scelto di non scaricare anche questo carico su di loro.

Una criticità emersa soprattutto in fase di definizione della vendita di un prodotto GPOne da parte dei commerciali riguarda la difficoltà nel definire cosa dare al cliente per dargli una determinata feature.

La precedente struttura del file di licenza di GPOne prevedeva un struttura a due livelli. Come si può vedere in figura 1.2, il primo livello prevedeva le applicazioni, che in linea teorica dovevano poter essere standalone e che dovevano anche mettere un limite al numero di utenti o di pannelli da officina utilizzabili, e per ogni applicazione potevano esserci uno o più moduli che abilitavano una serie di features o voci di menu. La figura non include tutti i moduli e le applicazioni effettive che sono presenti su GPOne, ma solo quelle necessarie per la spiegazione. Una suddivisione di questo tipo è stata ereditata da GP9Over, tuttavia su GPOne questa suddivisione ha creato non pochi

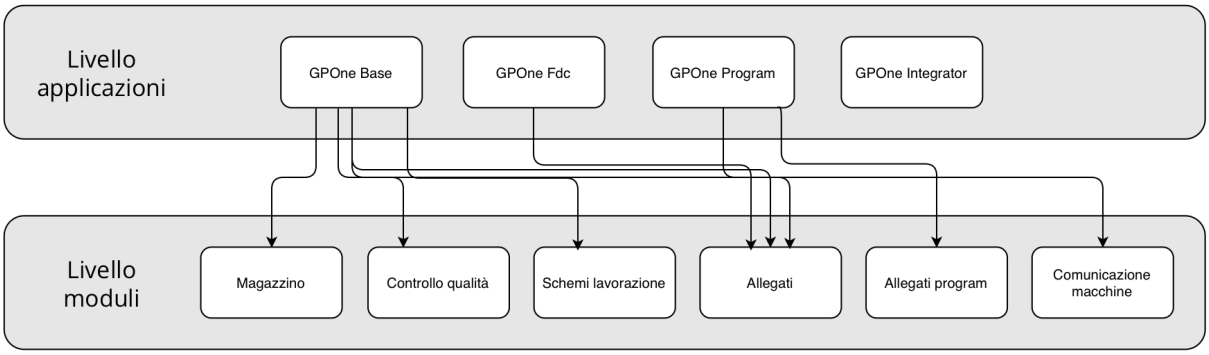

Figura 1.2: Struttura vecchio file licenza

problemi.

L'assunto che era stato fatto nelle prime analisi di sviluppo del prodotto, ovvero che si doveva avere una serie di applicazioni slegate, con il passare degli anni e degli sviluppi è venuto meno. Prendendo sempre come riferimento la figura 1.2, sarebbe logico pensare che le quattro voci nel livello applicazioni possano funzionare in modo indipendente; tuttavia tutte le applicazioni devono prevedere la presenza di GPOne Base per poter avere anche solo l'installazione dei servizi core. Più volte è capitato che un commerciale facesse pervenire l'ordine cliente con l'assenza del numero di utenti sul Base o senza addirittura GPOne Base; come conseguenza di ciò, il consulente andato da cliente ad installare si ritrovava con un'installazione fallita e la Ricerca e Sviluppo con un ticket in più. Al netto delle mancanze di attenzione e di comunicazione, una struttura del genere non rispecchiava più la struttura interna di GPOne, dunque non era più accettabile fornire questo tipo di informazioni al reparto Commerciale.

Ultima motivazione ma non meno importante, il flusso di rilascio licenza o di aggiornamento licenza è ancora troppo dipendente dal vecchio flusso con cui si gestivano le licenze di GP9Over. Si è accennato al fatto di come un commerciale, una volta effettuata la vendita, passi il contratto di vendita a un dipendente del Backoffice e questi inserisca un ordine cliente all'interno del gestionale aziendale. Il gestionale aziendale è GP9Over e verrà approfondito in una sezione dedicata in seguito. L'ordine cliente è un documento vincolante che contiene tutte le informazioni della vendita, chi ha effettuato la vendita, i dati del cliente, come indirizzo e ragione sociale, la data di acquisto e l'eventuale data di consegna, i prodotti acquistati con le relative quantità e il prezzo di acquisto.

Una volta inserito l'ordine cliente, partono una serie di automazioni che vanno a censire una

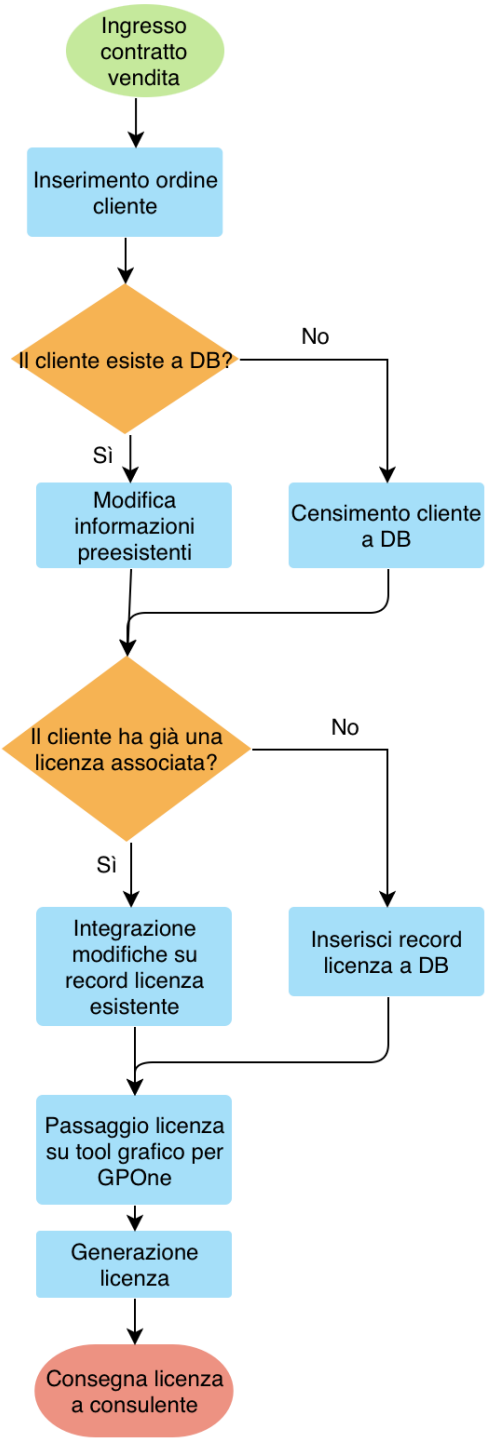

Figura 1.3: Vecchio flusso di inserimento licenza

serie di informazioni nelle tabelle del gestionale. Innanzitutto, vengono censite le informazioni nella tabella Clienti, qualora non fossero già presenti. Dopodiché si passa all'inserimento di un record nella tabella Licenze collegata a quel cliente. Anche qui, nel caso il cliente avesse già una licenza associata per un dato prodotto, si vanno a integrare le informazioni sul record esistente. Il record di licenza come si è già accennato, contiene le informazioni delle applicazioni e dei moduli oltre che alla versione del software, quindi si andranno a censire anche le varie tabelle di collegamento. Raggiunto questo passo, interviene l'assistenza. Nell'assistenza è presente una sola persona con le conoscenze per gestire le licenze su GP90Over, quindi ogni inserimento ad hoc o modifica deve passare da lei; questa conoscenza tra i vari reparti è condivisa da pochissime persone, dunque già questo crea tendenzialmente un collo di bottiglia. Infine, per creare la licenza finale di GPOne, viene utilizzato il tool grafico integrato con GP9Over mostrato in figura 1.4. Siccome il flusso era preesistente, la Ricerca e Sviluppo non aveva un modo agevole di modificare le licenze o di generarne una alla bisogna qualora si presentasse una delle problematiche citate in precedenza.

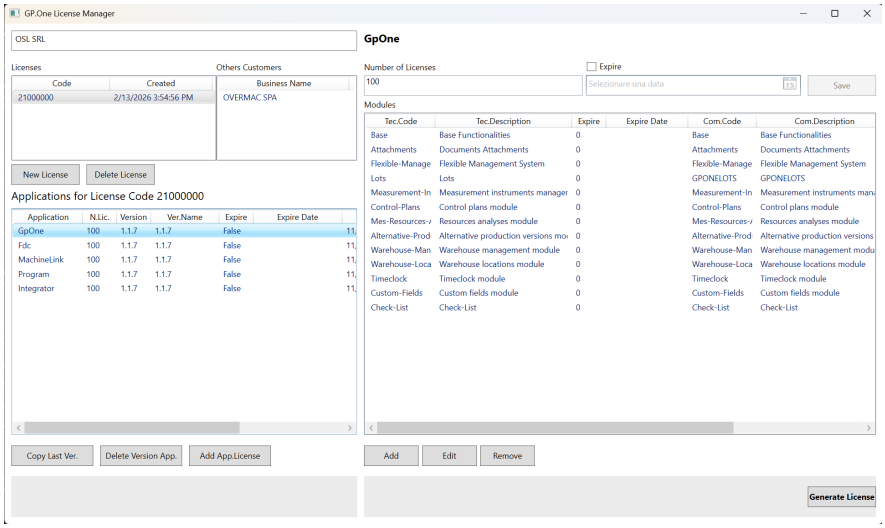

Figura 1.4: Vecchio tool di gestione licenza

1.3 Analisi dei requisiti dell'applicazione

Analizzate le inefficienze e i problemi che erano presenti, ci si concentrerà ora sul delineare quelli che sono stati gli obiettivi e i requisiti posti dagli stakeholders coinvolti.

Prima di tutto si identificheranno gli stakeholders. Come si è intuito dalle precedenti sezioni, lo sviluppo di questo applicativo andrà a impattare l'efficienza di una buona fetta di dipendenti OSL.

La lista degli stakeholders coinvolti:

- gli sviluppatori sia della Ricerca e Sviluppo che della Programmazione, in quanto sono direttamente responsabili dello sviluppo e del mantenimento futuro dell'applicazione;
- il management, inteso come l'insieme di figure rilevanti all'interno dell'azienda, tra cui il vice-presidente, il CFO, il responsabile di prodotto di GPOne, che coincide con quello di GP9Over in questo caso, che possono interfacciarsi con l'applicativo per richiedere un analisi dati tramite report;
- i consulenti e l'assistenza, che possono utilizzare l'applicativo per modificare o ottenere la licenza in caso di installazione presso cliente;
- analisti funzionali e commerciali, che si occuperanno di inserire e validare la licenza.

1.3.1 Requisiti funzionali

La raccolta dei requisiti è stata effettuata nei primi mesi del 2025, la maggior parte da Gennaio a Marzo. Di seguito si analizzeranno in maniera più dettagliata i requisiti funzionali presenti in tabella 1.1.

ID	Requisito	Descrizione
RF01	Gestione licenze per cliente	Il sistema deve permettere la gestione delle licenze per cliente.
RF02	Autenticazione con mail aziendale	Dev'essere implementata un'autenticazione integrata tramite la mail aziendale.
RF03	Gestione ruoli e permessi	Dev'essere implementata una gestione dei permessi per gli utenti che si interfacceranno all'applicazione.
RF04	Integrazione con GP9Over	Dev'essere presente l'integrazione bidirezionale tra l'applicazione e GP9Over.
RF05	Normalizzazione struttura licenza	La struttura delle licenze dev'essere normalizzata a un unico livello.
RF06	Tracciamento delle modifiche	Qualsiasi operazione di CRUD effettuata sulle licenze dev'essere registrata.
RF07	Raggiungibilità senza VPN	L'applicazione dev'essere raggiungibile anche non essendo connessi alla rete aziendale.
RF08	Export dati/report	Dev'essere possibile accedere ai dati dell'applicazione per consentirne l'analisi dati.
RF09	Cifratura licenza	Dev'essere modificata la cifratura del file di licenza.
RF10	Comunicazione GPOne con applicazione online	GPOne deve poter interagire con l'applicazione licenze da remoto.
RF11	Gestione cambiamenti licenza su GPOne	Dev'essere gestito come GPOne si comporta all'aggiornamento del file di licenza.
RF12	Gestione online/offline installazione GPOne	Il setup di GPOne deve poter funzionare se il server cliente non è online.

Tabella 1.1: Tabella dei requisiti funzionali del sistema

Il requisito RF01 seppure molto semplice, racchiude il core delle funzionalità principali che l'applicazione delle licenze deve avere: nella prima iterazione si devono poter gestire i clienti, i moduli, le versioni delle licenze e le licenze stesse. Questo prevede lo sviluppo di una API che permetta tutte le operazioni di CRUD (creazione, lettura, modifica, cancellazione) per tutte le entità che abbiamo elencato. Inoltre sono state richieste anche una serie di interazioni più particolari per la gestione delle licenze, come una chiamata di blocco licenza o di approvazione licenza; il blocco serve in tutti quei casi in cui il cliente è insolvente, mentre l'approvazione è un passaggio intermedio che serve a validare l'inserimento di una licenza, e questo può essere fatto dagli analisti una volta che viene validato l'insieme dei moduli venduti. Una volta approvata la licenza, questa ottiene una product key e può anche essere scaricata fisicamente sotto forma di file cifrato.

Al primo requisito sono intrinsecamente legati anche i requisiti RF02 e RF03. Chiaramente è necessaria una forma di autenticazione, idealmente tramite mail aziendale. Le mail aziendali sono gestite da Microsoft, dove è presente un tenant per l'azienda. Tramite questo tenant, i vari reparti accedono ai vari servizi tra cui Sharepoint, Microsoft Teams e Azure Devops. È fondamentale inoltre fornire permessi diversi ai vari utenti che si interfacceranno all'applicazione. Alcuni componenti della Ricerca e Sviluppo, specialmente chi sviluppa l'applicazione, devono sostanzialmente avere poteri illimitati. Inoltre, solo questa categoria di utenti può inserire e modificare un modulo o una versione, dato che passa direttamente dal flusso di rilascio di una nuova versione di GPOne. I commerciali e gli analisti devono poter inserire e modificare le licenze e i clienti, inoltre sono la categoria che può mettere il veto sull'approvazione di una licenza, e dunque devono poter approvare una licenza. L'assistenza può inserire e modificare una licenza e in caso di necessità può anche bloccarla. I consulenti possono vedere le licenze per un cliente specifico, possono ottenere la chiave di attivazione o il file fisico della licenza, ma non devono poter modificare le licenze in alcun modo.

Il requisito RF04 è fondamentale per l'integrazione con i flussi aziendali attuali. È necessario che sia presente una chiamata API esposta richiamabile da GP9Over nel momento in cui un ordine cliente genera una licenza. Questa chiamata deve andare a censire il cliente e a inserire le informazioni della licenza in stato da approvare. È chiaro che il processo deve funzionare anche nella direzione opposta: qualora ci dovessero essere modifiche alla licenza sul applicazione, questa deve comunicare a GP9Over le modifiche effettuate.

Il requisito RF05 riguarda l'organizzazione dei moduli nell'applicazione licenze e su GPOne.

Come già si è visto anche in figura 1.2, l'evoluzione della struttura sta nel collasso dei due livelli in un singolo livello, in cui ci saranno solo moduli. Questo permetterà una gestione molto più lineare e snella sia per quanto riguarda il censimento dei moduli stessi, che della creazione della licenza. Renderà inoltre più semplice anche la gestione delle date di scadenza di ogni singolo modulo.

Il requisito RF06 serve a scopo di audit. Si ipotizzi una modifica a una licenza, magari un blocco di una licenza che non era stato concordato con il reparto commerciale, e ciò ha portato a un blocco dell'applicazione di GPOne presso il cliente, con conseguente apertura di ticket in assistenza e chiamate aggressive. Il tracciamento di tutte le modifiche permetterebbe di risalire all'utente che ha compiuto l'azione. Chiaramente il tracciamento si può estendere anche a tutte le chiamate effettuate all'API, ma questo punto verrà affrontato nei requisiti non funzionali.

Il requisito RF07 richiede che l'applicazione sia esposta anche al di fuori della rete aziendale. Questa richiesta deriva in parte da chi si reca presso cliente. Un esposizione dell'applicazione anche da reti clienti permetterebbe ai consulenti di autogestirsi per l'ottenimento della licenza. La Ricerca e Sviluppo tuttavia ha messo questo requisito anche e soprattutto per altri due motivi. Uno è la comunicazione da remoto, che si lega ai requisiti RF10 e RF12. L'altro e forse più importante è che l'applicazione potrebbe essere la prima volta che l'azienda si interfaccia allo sviluppo in cloud. L'azienda, come già detto nelle sezioni precedenti, ha sempre fatto delle installazioni on-premises il suo punto di forza. Questo però ha portato una mancanza di competenze nello sviluppo e nell'utilizzo dei servizi in cloud sempre più grande e di conseguenza anche uno scetticismo diffuso nei dipendenti e negli sviluppatori più anziani nell'effettuare questo tipo di sviluppi. Anche il management è stato convinto che una mancanza di questo tipo di competenze in una software house moderna non sia più accettabile.

Il requisito RF08 è stato richiesto per fare analisi dei dati sulle tabelle dell'applicazione. Il management e gli analisti possono voler controllare quali installazioni di GPOne sono ancora attive presso i clienti, quali sono i clienti che hanno acquistato il maggior numero di moduli, quali sono i moduli più venduti. Da queste estrazioni dati gli analisti possono poi crearsi dei report personalizzati o delle integrazioni con PowerBI. La richiesta in questa prima iterazione non prevede la generazione automatica di un report dall'applicazione ne di impostare una dashboard, semplicemente di dare accesso in lettura al database.

Il requisito RF09 è semplice. Il metodo di cifratura utilizzato non è più accettabile per i motivi

elencati nella sezione precedente. Il nuovo metodo deve essere svincolato da eventuali chiavi calcolate dalla macchina di installazione, almeno per installazioni che permettono la comunicazione con un servizio esterno. La direzione scelta porta all'utilizzo di una chiave pubblica per la cifratura e una chiave privata che applichi una firma digitale alla licenza, in modo da non essere modificabile da esterni senza conoscere la chiave privata, ma consentendone la lettura alle applicazioni che ce l'hanno. Così facendo il software diventa resistente anche alle migrazioni di macchine virtuali. Per le installazioni offline va comunque rivisto il metodo di calcolo della chiave sul server cliente, in modo da generare una chiave il più possibile svincolata da migrazioni di macchine virtuali sullo stesso server.

Gli ultimi 3 requisiti, RF10, RF11 e RF12 riguardano tutti modifiche da portare sull'applicazione di GPOne. Il requisito RF10 riguarda il modo di comunicare tra GPOne e l'applicazione licenze. È immediato pensare che vada esposta una chiamata API per aggiornare le informazioni di licenza che l'installazione di GPOne ha in un dato momento, viceversa ci si può salvare un'informazione lato applicazione online che identifichi l'ultimo dispositivo da cui è stata effettuata la chiamata. In questo modo si può anche tenere traccia qualora ci sia un cambiamento non previsto del server.

Il requisito RF11 riguarda il modo in cui GPOne reagisce al cambiamento della licenza a runtime. Con la vecchia gestione della licenza non era previsto che la licenza venisse aggiornata durante il funzionamento dei servizi. Con l'implementazione della nuova gestione, andranno implementate nei vari servizi delle routine di aggiornamento delle informazioni contenute in memoria in quel momento. Inoltre, è necessario anche implementare un metodo di refresh forzato della licenza, qualora venga attivato un modulo e il cliente volesse utilizzarlo subito; questo metodo dev'essere qualcosa di eseguibile anche da un utente non troppo avanzato, all'interno di GPOne stesso. Ovviamente le modifiche del requisito RF11 riguardano solo il caso in cui venga fornita la connessione tra il server GPOne e l'applicazione licenze.

Il requisito RF12 estende il discorso del precedente requisito anche alle installazioni offline. Nel caso di un'installazione offline, il setup già usato deve continuare a funzionare, quindi se fornito un file di licenza fisico l'installazione deve essere eseguita senza attivare i controlli online. Nel caso di un installazione online, il setup deve poter scaricare il file tramite l'utilizzo di una product key e solo in questo caso deve attivare i controlli periodici di validità della licenza.

1.3.2 Requisiti non funzionali

Affrontati i requisiti funzionali, si elencheranno i requisiti non funzionali. Tali requisiti, elencati in tabella 1.2 riguardano solo l'applicazione delle licenze e derivano da analisi fatte principalmente tra i reparti coinvolti nello sviluppo effettivo dell'applicazione e il responsabile di prodotto.

ID	Requisito	Descrizione
RNF01	Interfaccia intuitiva	Il sistema deve essere semplice e intuitivo.
RNF02	Predisposizione al supporto multilingua	Dev'essere predisposta almeno la traduzione delle lingue italiano e inglese.
RNF03	Utilizzo di un database relazionale	I dati devono essere salvati in un database relazionale.
RNF04	Scalabilità ed evoluzione	Il sistema dev'essere predisposto per aggiungere facilmente nuove funzionalità.
RNF05	Prestazioni ed disponibilità	Il sistema dev'essere raggiungibile e in grado di fronteggiare il carico richiesto.

Tabella 1.2: Tabella dei requisiti non funzionali del sistema

Il requisito RNF01 non ha bisogno di troppe spiegazioni. Nelle applicazioni moderne non è più accettabile trascurare i concetti fondamentali di esperienza utente e di usabilità, anche se l'applicativo viene utilizzato solo internamente. Un prodotto intuitivo e facilmente usabile può ridurre il tempo che un dipendente impiega per eseguire un dato compito, oltre a renderlo più piacevole se viene coniugato a una UI pensata nel modo giusto.

Il requisito RNF02 riguarda il supporto in più lingue dell'applicazione. Le applicazioni moderne devono avere quantomeno la localizzazione dell'interfaccia in lingua inglese. Essendo OSL un'azienda italiana e dato che la quasi totalità dei dipendenti è italiana, è scontato che debba esserci anche la localizzazione in lingua italiana. Predisponendo tuttavia la localizzazione in più lingue, si rende possibile se necessario in futuro estendere le traduzioni a nuove lingue. Per OSL non è una possibilità così remota, dato che un rivenditore storico di GP9Over si interfaccia con il mercato spagnolo, e Overmach vende macchinari anche nel territorio africano, in paesi a lingua prevalentemente francese.

Il requisito RNF03 tratta di una richiesta specifica avanzata dal responsabile di prodotto. L'azienda ha sempre lavorato con SQL Server su GP9Over, dunque la maggior parte dei dipendenti è capace di comprendere e di interfacciarsi con le logiche di un database relazionale. La ricerca e sviluppo non è nuova all'utilizzo di database non relazionali come MongoDB, tuttavia l'utilizzo di quel tipo di tecnologia richiederebbe degli sviluppi in più per l'estrazione di dati, oltre che una formazione di molte persone all'interno dell'azienda.

Infine, i requisiti RNF04 e RNF05 riguardano l'architettura e l'implementazione della soluzione. Essendo una prima iterazione dello sviluppo, nei mesi e negli anni successivi saranno richieste nuove funzionalità per raffinare il prodotto. Addirittura si potrebbe voler spostare tutta la gestione interna dell'azienda su questo software qualora si decidesse di dismettere GP9Over. L'architettura della soluzione deve permettere uno sviluppo agevole per chiunque si interfacci con la codebase. Inoltre essendo esposto a chiamate da clienti esterni, il software dev'essere raggiungibile per una buona percentuale di tempo nel corso dell'anno. Il carico chiaramente non è quello che ci si può aspettare da un SaaS che ha il traffico di milioni di utenti, ma come si può prendere come limite superiore il numero di clienti attuali che ha OSL. Ovviamente il discorso sulle prestazioni vale anche e soprattutto per gli interni, quindi l'applicazione non deve avere tempi di risposta e di rendering proibitivi.

1.4 Contesto normativo

Sezione a parte meritano le analisi relative alle normative vigenti in ambito informatico nel momento dello sviluppo dell'applicativo. Non verrà trattato il CRA (Cyber Resiliency Act), che nonostante sia entrato in vigore a dicembre 2024, fornisce alle aziende 36 mesi di tempo per compiere gli adeguamenti necessari. OSL ha preso atto delle normative del momento e ha deciso di posticipare la messa in atto delle modifiche previste dal CRA nella seconda metà del 2026, ben lontano dal termine ultimo previsto del 11 dicembre 2027.

Le normative che si prenderanno in esame sono due: GDPR e NIS2.

1.4.1 GDPR

Il GDPR (General data protection regulation, regolamento (UE) n. 2016/679) è un regolamento dell'Unione Europea che è stato pubblicato sulla Gazzetta Ufficiale dell'Unione Europea ed è operativo dal 25 maggio 2018. Tale regolamento si occupa del trattamento dei dati personali e della privacy e l'UE tramite questa normativa punta a rafforzare la protezione dei dati personali dei suoi cittadini e dei suoi residenti, sia all'interno che all'esterno dell'Unione Europea, fornendo quindi ai proprietari dei dati personali il controllo su di essi. Questo ha di fatto semplificato il contesto normativo dei paesi membri, fornendo un regolamento uniformato per ognuno di essi. Per dato personale s'intende qualsiasi informazione che possa identificare una persona, questa informazione può essere diretta (nome, cognome, email, ...) o indiretta (indirizzo IP della connessione, id del dispositivo, ...). Qualora un azienda non sia a norma con il regolamento, quest'ultima rischia sanzioni che possono arrivare fino a 20 milioni di euro annui o al 4% del fatturato annuo. Il GDPR si fonda su una serie di principi chiave per garantire la conformità di un'azienda, elencati nella tabella 1.3.

ID	Principio	Descrizione
G1	Legalità, equità, trasparenza	L'utente deve sapere quali dati vengono raccolti e perché.
G2	Limitazione dello scopo	I dati devono essere raccolti per scopi specifici, espliciti e legittimi.
G3	Minimizzazione dei dati	Devono essere raccolti solo i dati necessari allo scopo.
G4	Precisione	I dati raccolti devono essere corretti.
G5	Limitazione dello stoccaggio	I dati non devono essere conservati per più tempo del necessario.
G6	Integrità e riservatezza	I dati devono essere protetti.
G7	Responsabilità	L'azienda deve poter dimostrare la propria conformità.

Tabella 1.3: Tabella dei sette principi chiave del GDPR

L'utente finale può esercitare una serie di diritti, espressi nei vari articoli del regolamento:

- diritto di essere informato, l'utente può richiedere informazioni relative ai suoi dati personali che l'azienda ha raccolto;

- diritto di accesso, l'utente può chiedere accesso ai suoi dati personali;
- diritto di rettifica, l'utente può richiedere la correzione dei suoi dati qualora siano sbagliati;
- diritto di cancellazione, l'utente può richiedere la cancellazione dei suoi dati;
- diritto di limitazione trattamento, l'utente può richiedere una limitazione del trattamento dei suoi dati;
- diritto di notifica, l'utente ha diritto a ricevere un avviso qualora suoi suoi dati vengano compiute delle azioni;
- diritto di portabilità, l'utente può richiedere che questi dati gli vengano forniti in un formato leggibile;
- diritto di opposizione, l'utente può opporsi al trattamento dei suoi dati in qualsiasi momento.

Non si entrerà nel dettaglio dei punti strettamente legali del GDPR. OSL dall'uscita della normativa si è adeguata al rispetto di queste regole. Dal 2018 per le vendite dei prodotti, oltre al contratto di acquisto, viene anche fatta firmare un'informativa sulla privacy e sul trattamento dei dati. OSL mantiene tutti i suoi dati sui dipendenti, clienti e fornitori all'interno dei server aziendali. Inoltre, i dipendenti vengono aggiornati su tale normativa con dei corsi di formazione nel momento del loro inserimento in azienda e ogni volta che entrano in vigore nuove normative o modifiche alle normative esistenti sull'argomento.

Per lo sviluppo dell'applicazione delle licenze, oltre a quanto già detto, ci si concentrerà più su alcuni punti più specifici relativi allo sviluppo. Tra i vari punti del GDPR, uno dei più collegati allo progettazione software riguarda il principio di "Privacy by design", contenuto nell'articolo 25 del regolamento. L'obiettivo di questo principio è far si che la privacy e la protezione dei dati non venga gestita solo a posteriori, ma che venga integrata già dall'inizio delle analisi e della progettazione del software, già in fase di raccolta dei requisiti e di selezione delle tecnologie e di conseguenza a partire dalla prima riga di codice che viene scritta. La scelta delle tecnologie per l'applicazione delle licenze è fondamentale, dato che essendo un'applicazione esposta, bisognerà usare uno stack sicuro e provider di servizi certificati, soprattutto nel caso in cui i dati non risiedano più all'interno del perimetro aziendale.

1.4.2 NIS2

La NIS2 (Network and Information Security 2) è una direttiva europea di recente pubblicazione(Direttiva UE 2022/2555). Pubblicata dall'UE nel 2022 e poi entrata in vigore per tutti gli stati membri dal 16 ottobre 2024, ha lo scopo di potenziare il livello di sicurezza informatica degli stati e delle loro infrastrutture, garantendone quindi la continuità di business in caso di guasti o attacchi informatici. La NIS2 è un'estensione della precedente NIS, che coinvolgeva solo una parte di aziende facente parte dei settori "critici". Dal momento in cui è andata in vigore la NIS2, la precedente NIS è stata dismessa.

La precedente NIS andava seguita da tutte quelle imprese che fornivano servizi critici alle infrastrutture del paese, indipendentemente dalla grandezza e dal fatturato di queste aziende; tra queste quindi possiamo elencare:

- settore energetico;
- settore bancario;
- settore sanitario;
- settore dei trasporti;
- aziende impiegate nella gestione di acque potabili e reflue;
- settore bancario e aziende impiegate nei mercati finanziari;
- amministrazioni pubbliche;
- provider di infrastrutture digitali;
- settore aerospaziale.

La NIS2 estende il raggio d'azione della normativa anche ad altre aziende di settori definiti "importanti", inoltre ingloba anche tutte le grandi e medie imprese, ovvero quelle aziende con almeno 50 dipendenti o un fatturato annuo maggiore di 10 milioni di euro. L'inclusione si estende anche alle PMI (piccole medie imprese) nel caso in cui facciano parte di una catena di approvvigionamento di qualche azienda soggetta alla NIS2. I nuovi settori previsti sono:

- servizi postali e di corriere;
- elaborazione e distribuzione;
- trattamento dei rifiuti;
- fornitori digitali;
- aziende manifatturiere;
- industria chimica;
- industria alimentare.

Sia OSL che Overmach, per grandezza, fatturato e settore, sono dunque soggetti alla normativa NIS2. Al momento della stesura del presente elaborato, il reparto IT di Overmach, insieme con il reparto qualità di OSL, è impegnato nel raggiungimento dei requisiti richiesti per la registrazione al portale NIS2. La conformità, per direttive aziendali, si otterrà prima per Overmach, che è delle due l'azienda più critica, a inizio 2026, e a seguire nei mesi successivi anche per OSL. Anche in questo caso, come fatto per il GDPR, per tutti i dipendenti aziendali si stanno somministrando corsi di formazioni al fine di fornire alla totalità delle due aziende le conoscenze per muoversi nel campo della normativa.

La NIS2 si focalizza molto sulla gestione del rischio cyber delle imprese. Le imprese devono sviluppare procedure organizzative e misure di sicurezza per gestire tale rischio. All'atto pratico, nello sviluppo di software nel caso in esame, ciò si traduce in:

- gestione degli incidenti, quindi avere un'organizzazione strutturale sia per quanto riguarda l'eventuale software che all'interno dell'azienda tale da rilevare e gestire un eventuale attacco informatico;
- continuità di business, bisogna dunque avere un'infrastruttura software in grado di resistere a un evenutale attacco, continuando a funzionare o in alternativa di avere un downtime estremamente limitato;
- sicurezza della catena di approvvigionamento, già citata in precedenza, questo discorso si estende non solo alle catene di fornitori di aziende reali, ma più in astratto anche alle

varie dipendenze che un software può avere dall'utilizzo di alcune librerie o alle relative infrastrutture di hosting scelte.

- obbligo di notifica, nel caso di incidenti particolarmente significativi bisogna segnalare tempestivamente l'avvenuto evento alle autorità competenti, nel caso italiano all'ACN (Agenzia per la Cybersicurezza Nazionale);

- utilizzo di crittografia e pratiche di sviluppo sicuro, come in parte già visto anche per il GDPR, l'utilizzo di tecniche di crittografia e protocolli di comunicazione sicura, autenticazione a più fattori e controllo degli accessi e dei permessi.

La normativa introduce una serie di requisiti non banali da seguire. Per i discorsi relativi alla continuità di business, è fondamentale avere dei meccanismi di backup automatici, delle impostazioni di replica per i servizi e di firewall. Per quanto riguarda la reattività in caso di incidenti, sono necessari meccanismi di auditing e log, questi devono essere all'interno dell'applicazione ma anche nell'infrastruttura che ospita il software. Alcuni di questi punti sono stati già previsti nei requisiti funzionali elencati in precedenza in tabella 1.1. In particolare i requisiti RF02 (autenticazione), RF03 (gestione ruoli/permessi) ed RF06 (logs delle modifiche) rientrano nelle pratiche di sicurezza dell'ultimo punto della lista.

2. Architettura software e progettazione flussi

2.1 Concetti di architetture software

Per affrontare gli argomenti delle prossime sezioni, è bene introdurre alcuni concetti relativi alle architetture software utilizzate. Tra le applicazione che si prenderanno in esame sono presenti sia architetture monolitiche che distribuite.

2.1.1 Architetture monolitiche

L'architettura monolitica è la tipologia di architettura presente da più tempo in assoluto. La sua struttura è semplice da capire, tutta l'applicazione è eseguita da un unico processo. Ciò si traduce un unico progetto da modificare e compilare, un'unica applicazione da testare e da rilasciare. Inoltre essendo tutto in un unico processo, i vari servizi e le classi possono comunicare tra loro all'interno del processo senza l'ausilio di sistemi di messaggistica particolari.

Pur essendo semplice, con l'aumentare della complessità dei software questo tipo di architettura è diventato velocemente obsoleto per vari motivi:

- più aumenta la dimensione del progetto, più i vari servizi sono collegati tra loro, portando quindi a una serie di dipendenze tra servizi non più gestibile;
- all'aumentare delle linee di codice, aumentano drasticamente i tempi di compilazione e di rilascio, inoltre diventa molto più complicato e talvolta impossibile effettuare un refactor del codice;
- non è possibile rilasciare o scalare solo una parte dell'applicazione;
- è molto complicato cambiare una delle tecnologie utilizzate (ad esempio il motore di database utilizzato o il linguaggio di programmazione).

Nonostante l'obsolescenza in molti ambiti di questo tipo di architettura, esistono ancora casi in cui utilizzarla è una scelta corretta e conveniente. Se l'applicazione è semplice e non si prevede una

scalabilità di quest'ultima, se il team di sviluppo che ci lavora è composto da poche persone, allora lo sviluppo rimane ancora sostenibile.

Nel corso degli anni, per far fronte alle varie criticità del monolite classico, le architetture monolitiche si sono sviluppate in diverse forme. Quella che si vedrà ora e più avanti nell'elaborato è la cosiddetta Clean Architecture.

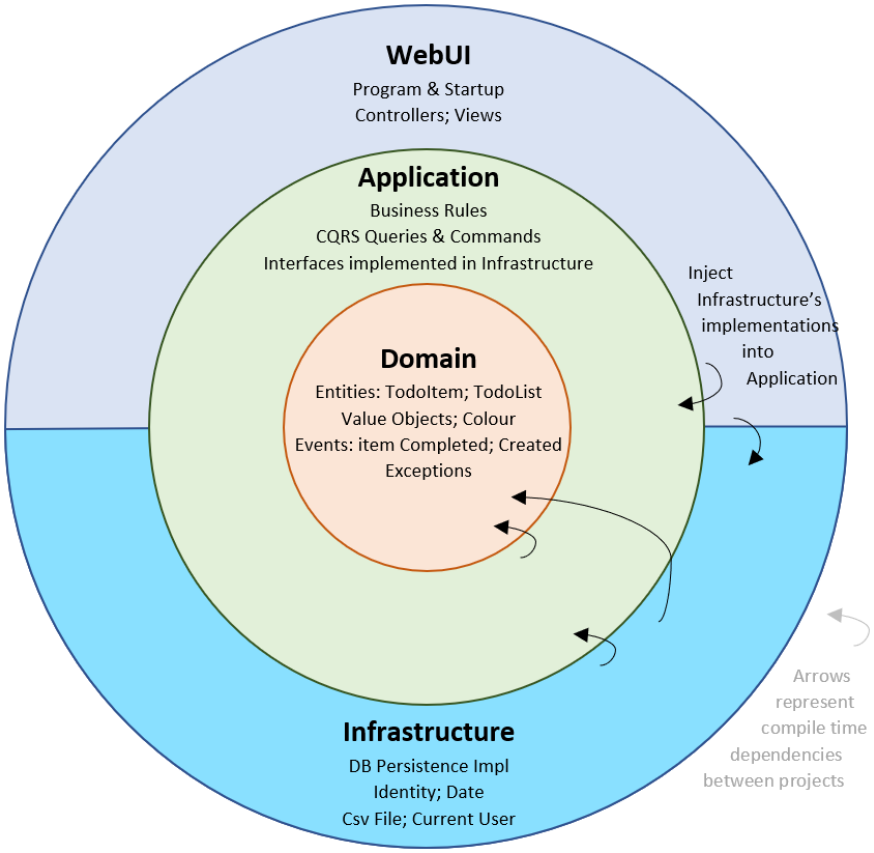

Figura 2.1: Schema della Clean Architecture di un'applicazione .NET (fonte blog.ndepend.com)

Quest'architettura è più strutturata rispetto ai primi monoliti, nonostante rimanga anch'essa monolitica. Il suo punto di forza è la separazione tra i vari strati della struttura, con le dipendenze che sono rivolte solo verso l'interno. Gli strati, come si può vedere in figura 2.1 sono quattro:

- domain, lo strato che comprende la definizione di tutte le classi di business, le eccezioni, le interfacce comuni;
- application, dove vengono definite le interfacce dei servizi che vengono implementati nello strato infrastructure e i servizi di business utilizzati;

- infrastructure, contiene tutta la parte di connettività ai servizi esterni, come i servizi di interfacciamento effettivo ai database;
- presentation (in figura 2.1 denominata WebUI), è lo strato che si interfaccia con l'utente, e contiene anche le varie inizializzazioni e dependency injection.

Questa suddivisione in strati permette al team di sviluppo di cambiare in modo semplice se ce ne fosse bisogno e senza ripercussioni sulla logica di business elementi come il database utilizzato o il framework usato nello sviluppo del frontend.

Il suo livello di astrazione è anche uno dei suoi difetti; per utilizzare al meglio questa scelta è necessario comprendere bene come funziona quest'architettura.

La clean architecture è una scelta valida qualora il team di sviluppo non sia troppo numeroso, sia discretamente competente sullo scrivere codice in questo modo.

2.1.2 Architetture distribuite

Con l'aumentare della complessità dei software monolitici, è stata necessaria un'evoluzione che andasse a spezzare lo sviluppo da un unico blocco di codice e un unico processo, a più processi sviluppati in modo indipendente tra loro. Da questa esigenza nascono le prime architetture distribuite.

Le applicazioni che seguono questa tipologia di sviluppo sono eseguite da processi indipendenti, talvolta anche su macchine diverse, con stack tecnologici possibilmente eterogenei tra i vari servizi. Ciò permette a ogni componente dell'applicazione di essere sviluppato in modo autonomo e di conseguenza è possibile gestire in modo separato la scalabilità e i rilasci di ogni componente. Potenzialmente è possibile avere un team di sviluppo separato per ogni componente.

Sebbene abbia dei benefici tangibili, le architetture distribuite si portano dietro una complessità molto più elevata. Seppur la complessità locale di ogni componente è molto minore rispetto a un monolite, il fatto che siano processi indipendenti impone un sistema di comunicazione esterno dei processi; inoltre, all'aumentare dei componenti dell'applicazione distribuita, aumenta la complessità della rete di comunicazione.

L'architettura distribuita più conosciuta e usata negli ultimi anni è l'architettura a microservizi.

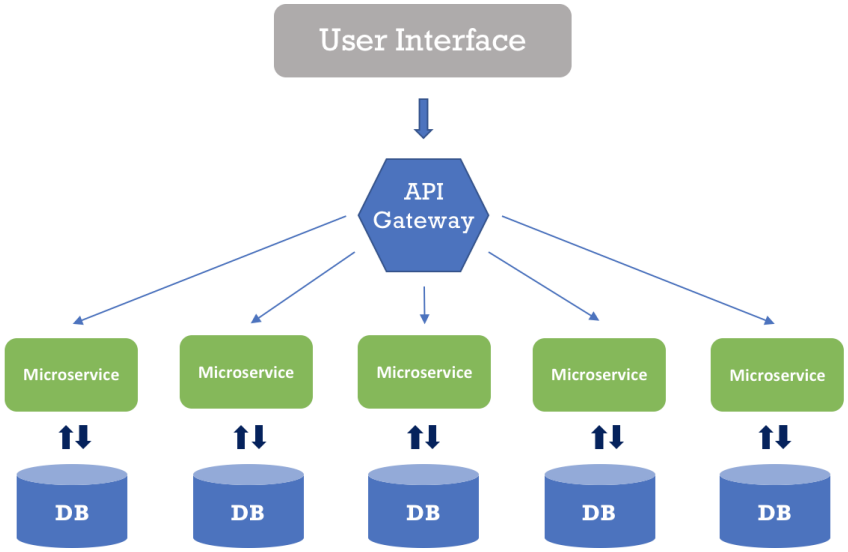

Figura 2.2: Schema di un'architettura a microservizi (fonte blog.nashtechglobal.com)

Questo tipo di architettura separa completamente i diversi servizi tra di loro; ogni componente ha in gestione una singola funzione aziendale, ha il suo dominio e il suo database. Nel caso di figura 2.2 le richieste lato client vengono gestite tramite un gateway API, che funge da punto d'ingresso in cui ogni richiesta viene indirizzata al servizio giusto.

Le sfide di questa scelta architetturale non sono di semplice risoluzione:

- il gateway API può essere un collo di bottiglia se dovessero esserci molte richieste in ingresso, causando rallentamenti nei microservizi coinvolti ed è pertanto necessario implementare dei meccanismi di load balancing ed eventualmente instanziare più servizi API;
- le transazioni che coinvolgono più database in sequenza necessitano di un'orchestrazione affinchè, in caso di errore, non rimangano delle incongruenze;
- eventuali debugging di funzioni che attraversano più microservizi sono più complessi e necessitano di una gestione di log e tracce più avanzata;
- le chiamate tra microservizi introducono una latenza non ignorabile.

Le comunicazioni tra microservizi possono essere affrontati in diversi modi; ogni servizio può esporre una API REST, per cui ogni richiesta viene fatta tramite una chiamata HTTP, o possono essere usati protocolli più complessi, come gRPC. Possono essere sfruttati sistemi più elaborati per

gestire la messaggistica tra i vari servizi. Qui entra in gioco un altro tipo di architettura distribuita, non necessariamente alternativa: l'architettura event-driven.

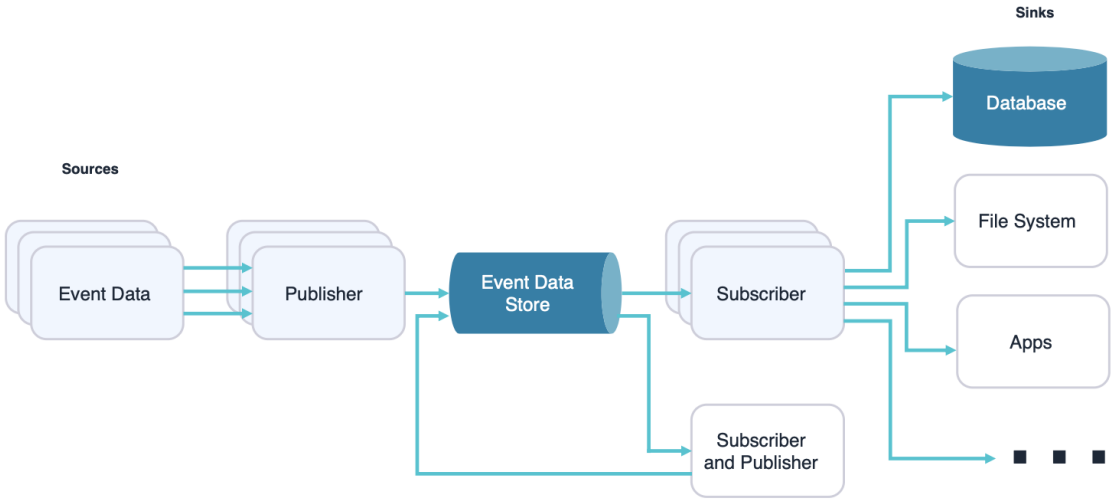

Figura 2.3: Organizzazione architettura event-driven (fonte hazelcast.com)

In questo caso cambia il paradigma di comunicazione tra servizi, che da richiesta/risposta diventa un reagire a un evento. Un evento può essere generato da un qualsiasi cambiamento di stato o un'azione eseguita da uno dei tanti servizi. In questo caso il servizio generatore viene definito publisher e i servizi che riceveranno tale evento vengono detti consumers. Come abbiamo detto questo sistema di comunicazione tra servizi genera latenza in quanto i servizi sono disaccoppiati e dunque viene introdotto dell'asincronismo nel sistema. Inoltre è necessario anche scegliere come impostare tale sistema: può essere centralizzato con un broker di messaggi, ovvero con un servizio che apre i canali di comunicazione ed è responsabile della gestione delle code di eventi, oppure è necessario avere una gestione su ogni servizio di dove mandare i messaggi e come riceverli (gestione brokerless). La gestione con un broker introduce un servizio in più da gestire, però rende molto più semplice la gestione già complicata degli eventi e inoltre, poichè il broker è centralizzato, se un servizio consumer dovesse andare in errore il messaggio non viene perso, ma viene tenuto in memoria dal broker, rendendo tutto il sistema più resiliente.

Le architetture distribuite hanno molti benefici ma altrettanta complessità; nello scegliere questo tipo di architettura è necessario un team di sviluppo molto preparato su questi argomenti complessi e che si preveda che l'applicazione scalerà molto in fretta; in caso contrario, conviene prendere in

considerazioni architetture più semplici.

2.2 Architettura e stack GP9Over

GP9Over è il software su cui l'azienda si è basata, e in parte si basa tuttora, sia per il fatturato della vendita software che per quello legato alla formazione fatta dai consulenti. Inizialmente chiamato GP90, prima dell'acquisizione da parte di Overmach, è stato sviluppato come necessità di un'azienda metalmeccanica nel 1991. Il software si è evoluto nel corso degli oltre 30 anni di servizio ed è tuttora manutenuto e aggiornato con feature minori. GP90 nel corso del tempo si è evoluto per gestire più parti della produzione dell'officina metalmeccanica tipo. Nel suo utilizzo più comune, GP9Over si può definire un MES(Manifacturing Execution System)-ERP(Enterprise Resource Planning), permettendo da una parte il monitoraggio in tempo reale delle lavorazioni effettive che vengono eseguite nell'officina, dall'altra tutta permette anche la gestione della pianificazione di tutte le lavorazioni, la gestione documentale, il carico sulle risorse, tutta la parte di MRP (Material Resource Planning) per la gestione del magazzino e delle giacenze. I punti di forza nel corso degli anni sono stati principalmente due:

- la capacità di andare incontro alle richieste cliente con la possibilità di realizzare personalizzazioni anche avanzate in tempo relativamente breve;
- la capacità di essere utilizzato per le perizie legate all'Industria 4.0.

Essendo un software di oltre 30 anni, purtroppo si porta dietro una serie di scelte a livello architetturale obbligate per gli anni in cui si è sviluppato. L'architettura come si può immaginare è la classica monolitica, in cui c'è una pesante parte di codice core che non è più possibile modificare. La scelta delle tecnologie è anch'essa dettata dall'età. La maggior parte della logica core è stata scritta in VB6 (Visual Basic 6), un linguaggio di programmazione ormai obsoleto e deprecato dal 2008 ufficialmente da Microsoft. Questo vincolo tecnologico obbliga gli sviluppatori ad dover eseguire l'editor e la compilazione in delle macchine virtuali con sistema operativo Windows XP, dato che la compilazione di VB6 non è più possibile in sistemi operativi più recenti di Windows 7, inoltre per delle librerie non supportate in GP9Over non è nemmeno possibile compilarlo in

Windows 7. Oltre ai problemi di compilazione e sviluppo, questo vincolo porta con se anche la difficoltà aumentata nel trovare programmatori che conoscano questo linguaggio.

GP9Over presenta anche un'altra parte di codice relativa a una funzionalità sviluppata in seguito intorno al 2010; la funzionalità chiamata "Raccolta Dati" serviva per eseguire un'applicazione su dei pannelli o dei palmari in officina per permettere agli operatori di registrare le lavorazioni o le operazioni che stavano eseguendo. Questa funzionalità è staccata dal monolite scritto in VB6 ed è stata realizzata in VB.NET. Pur cambiando la tecnologia, questa seconda applicazione è stata comunque sviluppata come un monolite. Il motore di database di GP9Over è SQL Server e ne supporta più versioni, a seconda della versione di GP9Over.

Dato l'ormai pluridecennale ciclo di vita di questo software, GP9Over permette l'integrazione con la maggior parte dei sistemi informativi gestionali presenti in commercio, come può essere il gestionale SAP, e attraverso varie modalità, che sia tramite database di frontiera, file CSV o anche semplici file di testo; tali integrazioni vengono ormai sviluppate con poco sforzo se non nullo.

La maggior parte delle modifiche che vengono richieste ad oggi riguardano piccole personalizzazioni grafiche o integrazioni. Tali modifiche possono essere fatte lato codice, qualora fosse una modifica un po' più corposa, tramite lo sviluppo e la compilazione di una DLL (dynamic linked library, una libreria di codice e funzioni condivisa), che viene poi rilasciata nel percorso d'installazione del cliente.

Tuttavia per far fronte all'enorme richiesta, è stato reso possibile anche effettuare personalizzazioni tramite SQL; in questo caso, se la personalizzazione è di facile realizzazione, è possibile addirittura che sia un consulente a realizzarla, abbassando di molto il tempo atteso dal cliente per ottenere la modifica. Chiaramente questo ha portato a un'anarchia sulle installazioni, con una serie di sviluppi completamente sregolati di stored procedures e soprattutto di trigger SQL, questi ultimi ormai sconsigliati da svariati anni per via della perdita di controllo che causavano sulla logica core.

Per lo scopo dell'elaborato, era necessario introdurre questo software dato che viene usato come gestionale interno e su cui risiedono tuttora le informazioni relativi ai clienti e alle licenze. Tuttavia non si entrerà nei dettagli del codice dato che l'integrazione lato GP9Over è stata realizzata tramite una semplice chiamata REST eseguita periodicamente una volta al giorno all'endpoint dell'applicazione licenze ed è stato gestito dal reparto Programmazione GP9Over.

2.3 Architettura e stack GPOne

Nonostante GP9Over avesse fatto le fortune dell'azienda durante il periodo iniziale dell'industria 4.0, la grande quantità di vendite, installazioni e personalizzazione richieste fecero uscire tutti i limiti del gestionale non più adatto ad evolversi negli anni successivi. Con l'acquisto di OSL da parte di Overmach, si iniziò a delineare il progetto di riscrittura del gestionale in chiave moderna, GPOne. Overmach decise di instanziare circa 4 milioni di Euro per finanziare l'inizio delle analisi e dello sviluppo nel 2019, con l'obiettivo di ottenerne una prima versione vendibile tra il 2022 e il 2023. Le prime installazioni effettive del prodotto sono state effettuate nel 2023. Chiaramente l'obiettivo del prodotto non è di essere la copia esatta di GP9Over scritta con un linguaggio e un'interfaccia un po' più moderni, ma di andare ad implementare meglio le funzionalità del vecchio gestionale ed estendere con l'aggiunta di nuove funzionalità inedite.

L'architettura scelta per questo applicativo è quella a microservizi. Consci dei limiti e delle criticità del precedente monolite, si è cercato di andare il più possibile a separare in diversi servizi le varie funzionalità standalone del software. Il software si presenta come un appplicativo web rilasciato in questi primi anni di vita su server on-premises in loco dal cliente, con sistemi operativi Windows, ed è compatibile dalla versione 2019 di Windows Server e da Windows 10.

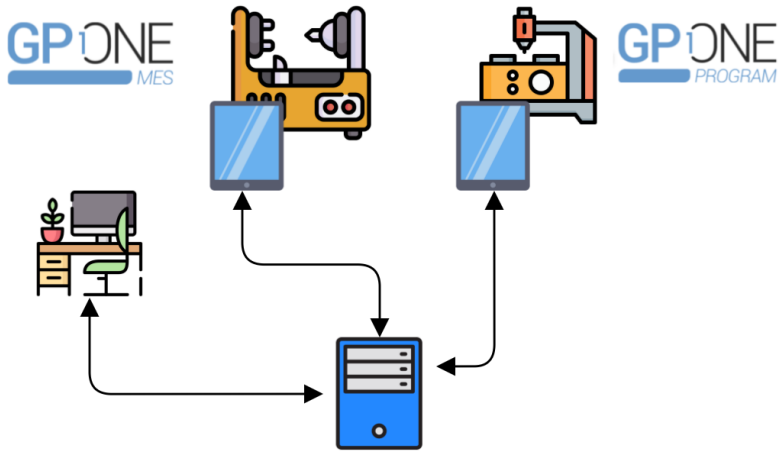

Figura 2.4: Schema di un'officina con GPOne (icone da Flaticon - utenti Handicon, Freepik, srip e DinosoftLabs)

Come si può vedere in figura 2.4, il software si inserisce nei vari processi di un'officina metalmeccanica.

È presente un'interfaccia per l'utilizzo da backoffice (figura 2.5). Da qui è possibile gestire tutta la parte relativa alle anagrafiche delle varie risorse aziendali e degli articoli, tutta la parte relativa alla pianificazione e al rilascio in produzione di commesse e fasi di lavoro, la gestione di magazzino e dei documenti di gestione merci e ordini cliente e fornitore. L'interfaccia si presenta come un classico gestionale, con la maggior parte delle pagine che sono griglie di tabelle e vari form per la gestione CRUD, oltre a qualche pagina più evoluta relativa al monitoraggio del carico e delle attività delle risorse e alla visualizzazione grafica delle distinte e degli alberi di produzione.

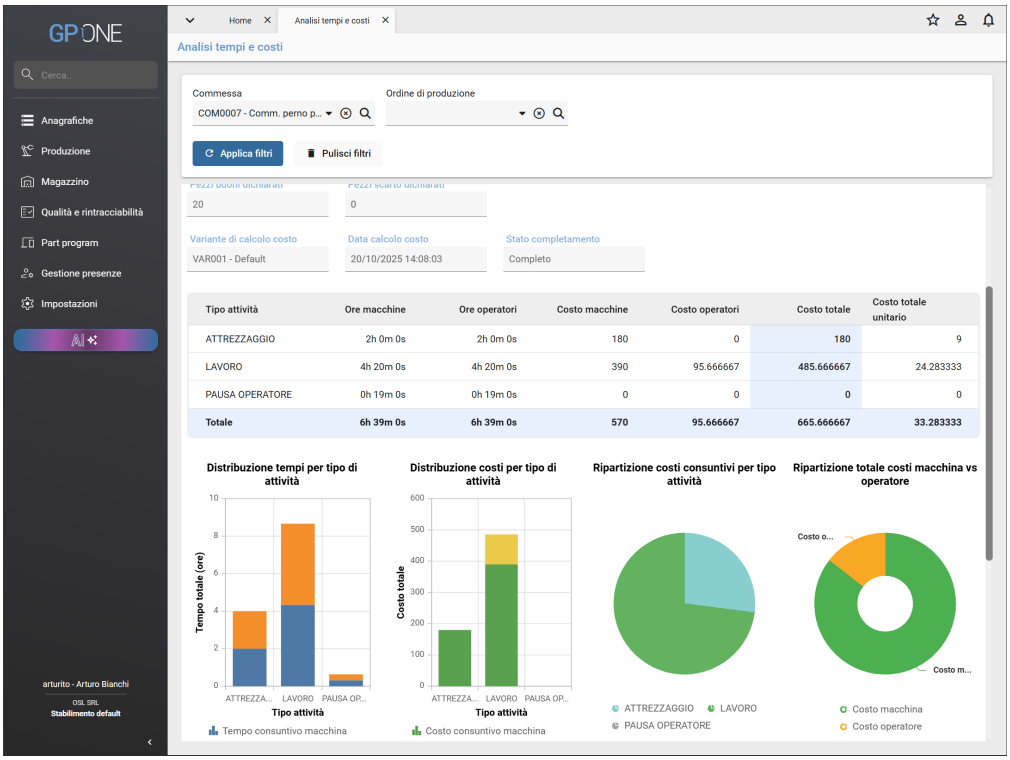

Figura 2.5: Interfaccia di GPOne MES

Un secondo client è presente per i pannelli e i tablet a bordo macchina (figura 2.6). Questi sono chiamati pannelli Fdc. Attraverso questi dispositivi edge gli operatori in officina possono registrare tutte le operazioni che stanno eseguendo, dall'inizio della produzione delle varie fasi di lavoro, alla dichiarazione dei pezzi lavorati, le dichiarazioni relative alla qualità dei semilavorati e la raccolta di misurazioni per i controlli relativi alle normative di qualità ISO/TS.

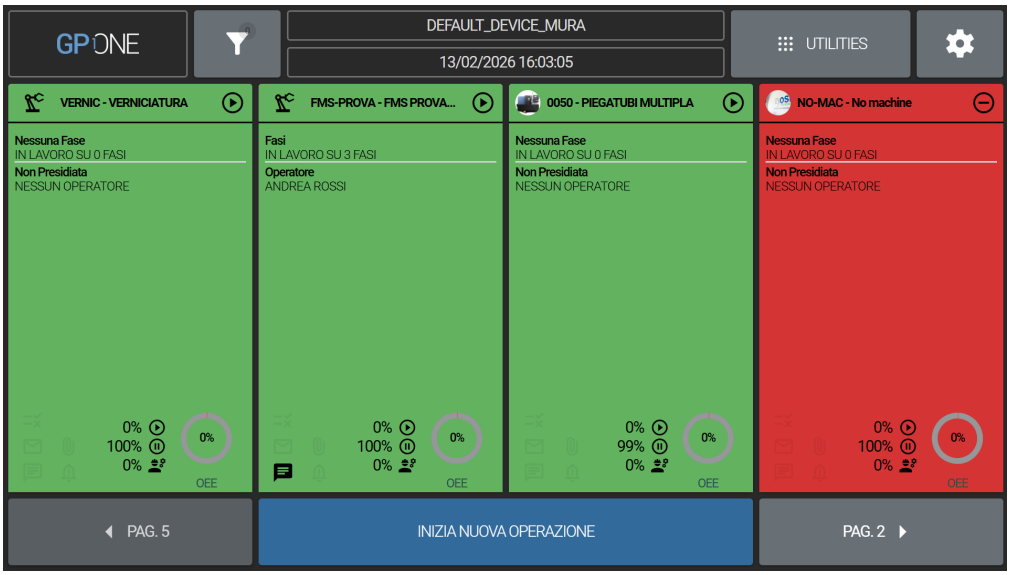

Figura 2.6: Interfaccia di GPOne FDC

Un ultimo client si occupa del trasferimento programmi macchina (figura 2.7). Questi sono sempre su dei pannelli a bordo macchina e sono chiamati pannelli Program. L'interfaccia è molto semplice dato che il suo obiettivo è di permettere all'operatore di trasferire dal server alla macchina e viceversa i vari programmi di lavorazione (file di testo scritti in linguaggio G-CODE) inseriti su GPOne dall'ufficio tecnico e di poterli modificare se necessario.

Tutti questi client si interfacciano con il server su cui sono installati i vari servizi. Ogni client si può interfacciare con uno o più servizi che sono hostati sulla macchina. Questi servizi infine si interfacciano con dei database SQL Server. La soluzione del software è suddivisa in una serie di repositories diversi:

- backend, che contiene la parte server del software, con oltre 40 progetti diversi;
- frontend, contenente la parte client dell'applicativo lato gestione backoffice e gestione produzione;
- totem, contiene la parte relativa al client dei pannelli Fdc;
- program, contiene la parte client relativa al Program.

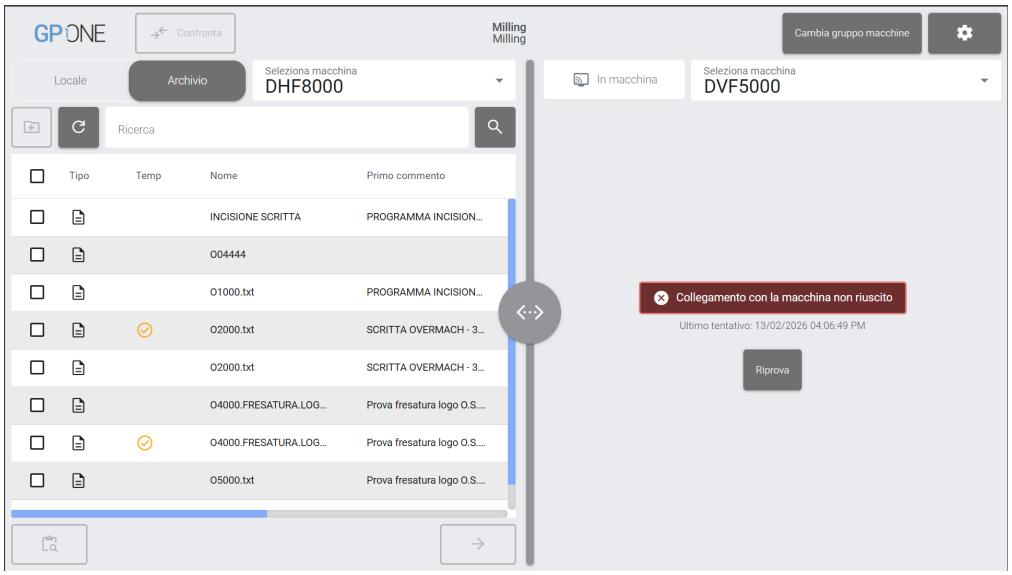

Figura 2.7: Interfaccia di GPOne Program

Sono presenti anche altri repositories contenenti l'implementazione di servizi più specifici per la gestione dei report o per le integrazioni con GP9Over, ma essendo principalmente dei plugins scollegati dalla licenza non si prenderanno in considerazione.

Tutte le soluzioni client sono state implementate sfruttando il framework di Angular, mentre la soluzione backend è stata realizzata in .NET di Microsoft e come linguaggio C#. Al momento della stesura dell'elaborato, le versioni utilizzate sono la v20 per Angular e .NET 8 per il backend, con in programma l'aggiornamento a .NET 10 e ad Angular v21 fissato ad Agosto 2026. Le regole della Ricerca e Sviluppo sull'aggiornamento .NET sono state quelle di aggiornare il framework nel giro di pochi mesi dall'uscita dell'ultima versione LTS (Long Term Support) che solitamente avviene una volta ogni 2 anni e questo perchè le versioni LTS hanno il supporto più lungo (36 mesi rispetto ai 24 mesi della Short Term Support STS).

Per affrontare i vari microservizi di GPOne bisogna introdurre un paio di tecnologie utilizzate.

2.3.1 GraphQL

GraphQL è un query language open-source per web API e si pone come alternativa alle classiche API REST. Sviluppato da Facebook nel 2012, è stato poi reso pubblico nel 2015. L'obiettivo è risolvere una serie di criticità presenti nelle normali API REST. Le migliorie più significative

rispetto a queste ultime sono:

- under/over-fetching, dato che GraphQL permettere di ottenere solo i dati che vengono richiesti dall'utente, senza la necessità di ritornare risposte con troppi dati o di dover fare più richieste per ottenere tutte le informazioni richieste dalla query;
- un unico endpoint, invece di avere tanti endpoint per ogni operazione, GraphQL espone un unico endpoint, tipicamente raggiungibile con /graphql, e tutte le richieste passano da li con una chiamata POST.

GraphQL utilizza uno schema come contratto client-server in cui sono definiti tutti i tipi di dato, le query, per la lettura, e le mutation, per la scrittura, utilizzabili. Sul server sono poi presenti dei resolver, delle funzioni che si occupano poi prendere il dato.

Codice 2.1: Esempio di richiesta/risposta GraphQL

```
#Richiesta
query GetUser {
  user(id: "123") {
    id
    username
    email
  }
}
#Risposta
{
  "data": {
    "user": {
      "id": "123",
      "username": "arturo_bia",
      "email": "arturobianchi@gmail.com"
    }
  }
}
```

2.3.2 Autenticazione JWT

Il JSON Web Token (JWT) è una stringa di caratteri codificata con algoritmo Base64 divisa in header, payload e signature. Il payload contiene l'algoritmo usato per applicare la firma digitale; il payload contiene tutta una serie di dati utente, come possono essere un id o una data di scadenza della validità del token; la signature è la firma digitale ottenuta dal server dalla combinazione tra header, payload e una chiave segreta del server e serve per controllare che il token non sia stato manomesso. Un token realizzato in questo modo permette al server che l'ha generato di non dover salvare in un proprio database l'informazione del token utente, rendendolo di fatto stateless. Il token viene salvato dal client in un cookie o in un LocalStorage. Una volta ottenuto il token, le richieste consecutive devono includere nell'header HTTP il token utilizzato. Questo può tornare molto utile qualora ci fossero più server che possono ricevere una richiesta, ad esempio un servizio API GraphQL che viene replicato in più istanze per necessità di alta disponibilità; in questo caso il token verrebbe ritenuto attendibile da qualsiasi istanza lo riceva grazie alla firma digitale, senza che nessuno dei server abbia salvato nessuna informazione del token. Nulla inoltre vieta di aggiungere altre informazioni al payload del token. In GPOne, oltre al token e all'id utente viene anche salvato il riferimento all'id dello stabilimento, che viene poi sfruttato subito dai resolver di GraphQL per poter filtrare adeguatamente i dati delle query.

2.3.3 RabbitMQ e MassTransit

Si è accennato nelle sezioni precedenti all'architettura event-driven e di come una delle categorie fosse quella basata su un broker, su un gestore degli eventi o dei messaggi. RabbitMQ è uno dei broker di messaggi più conosciuti e utilizzati. Il middleware è open-source e utilizza il protocollo di messaggistica AMQP (Advanced Messaging Queueing Protocol). Per l'utilizzo che ne viene fatto in GPOne, non si introdurranno le parti un po' più avanzate di RabbitMQ, come il suo utilizzo nei cluster.

Il concetto di publisher e consumer/subscriber di RabbitMQ è il medesimo introdotto nelle architetture event-driven. Il messaggio viene mandato a una struttura intermedia chiamata exchange e non viene dunque mandato direttamente alla coda del messaggio. In questo modo è possibile creare più di una coda per una tipologia di messaggio, l'exchange poi si legherà a queste code con

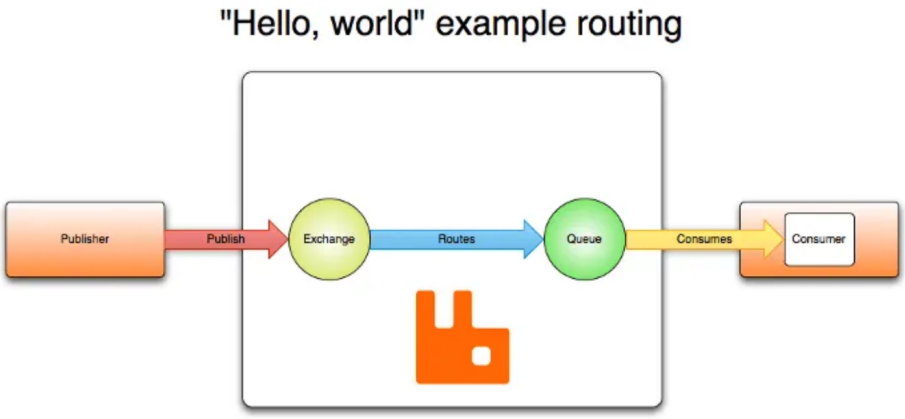

Figura 2.8: Esempio di semplice architettura RabbitMQ (fonte medium.com/@hoon33710/)

il binding. Inoltre grazie a questo elemento intermedio è possibile creare anche più di un consumer per una singola coda.

Esistono diversi tipi di exchange:

- fanout, il messaggio viene inviato a tutte le code legate all'exchange, in modo molto simile a una comunicazione broadcast;
- direct, il messaggio in questo caso ha una chiave di routing (o etichetta), e l'exchange instrada il messaggio verso le code che hanno la chiave di associazione esattamente uguale a quella del messaggio;
- topic, simile al direct ma più avanzato, in questo caso le etichette permettono la presenza di caratteri wildcards per instradare dinamicamente a più code i messaggi.

La potenza di RabbitMQ sta principalmente nella sua affidabilità, poichè se anche un messaggio viene inviato quando nessun consumer è in ascolto, il broker mantiene in memoria il messaggio finchè qualcuno non si metterà in ascolto sulla coda, oltre che alla sua capacità di disaccoppiamento tra publisher e consumer.

A livello implementativo dichiarare molte code ed exchange di RabbitMQ, oltre che gestire le connessioni al broker, può generare molte linee di codice ripetute. Masstransit è un framework per .NET che permette di astrarre questa gestione. Inizialmente open-source, è diventato a pagamento con l'uscita della versione 9. Implementando semplicemente le classi dei consumers e dei messaggi,

Masstransit instanzia direttamente le code e gli exchange; inoltre, con i servizi di bus che mette a disposizione, implementa anche meccanismi avanzati come il reinvio dei messaggi in caso di errore o timeout. Il framework ha un'integrazione avanzata con la dependency injection di .NET; inoltre permette di sostituire, cambiando semplicemente la registrazione dei servizi all'avvio, la tecnologia utilizzata al livello più basso per gestire i messaggi. Se si decidesse di sostituire RabbitMQ con Kafka, oppure si andasse in cloud con Azure Service Bus, la transizione sarebbe molto semplice e le modifiche a livello di codice sarebbero minime.

2.3.4 Entity Framework

Nelle applicazioni in .NET, nella maggior parte delle volte in cui ci si deve interfacciare con un database, sia esso relazionale o no, si utilizza la il framework di .NET Entity Framework (EF) Core. EF Core è un ORM (object-relational mapper) sviluppato da Microsoft, che permette di interagire con un database utilizzando solo il codice. Nel caso di un database SQL, il framework permette di evitare di scrivere comandi SQL.

Per creare e modificare le tabelle di un database EF Core utilizza delle classi POCO (Plain Old Clr Object) per crearsi un EDM (Entity Data Model), e basandosi su questo modello effettua query e crea le migrations che generano lo schema del database. Il framework sfrutta la libreria LINQ (Language Integrated Query) di .NET per eseguire delle query; la sintassi di LINQ viene poi tradotta a SQL nel momento di essere eseguita. È possibile anche eseguire una query SQL grezza qualora fosse necessario. Inoltre permette di gestire lato codice anche le transazioni e implementa un meccanismo di caching che entra in funziona quando una query viene ripetuta spesso. Infine, come per MassTransit, è possibile cambiare il motore di database sottostante senza stravolgere il codice, ma cambiando solo una stringa in dependency injection.

2.3.5 SignalR

Su GPOne ci sono più situazioni in cui è necessario che il server notifichi qualcosa ai client. Per risolvere questa necessità si utilizza SignalR, una libreria ASP.NET che permette di implementare funzionalità web real-time per eseguire il push di contenuti dal server ai client. Questo permette ai client di non dover eseguire chiamate polling al server. SignalR mette a disposizione un'API per le

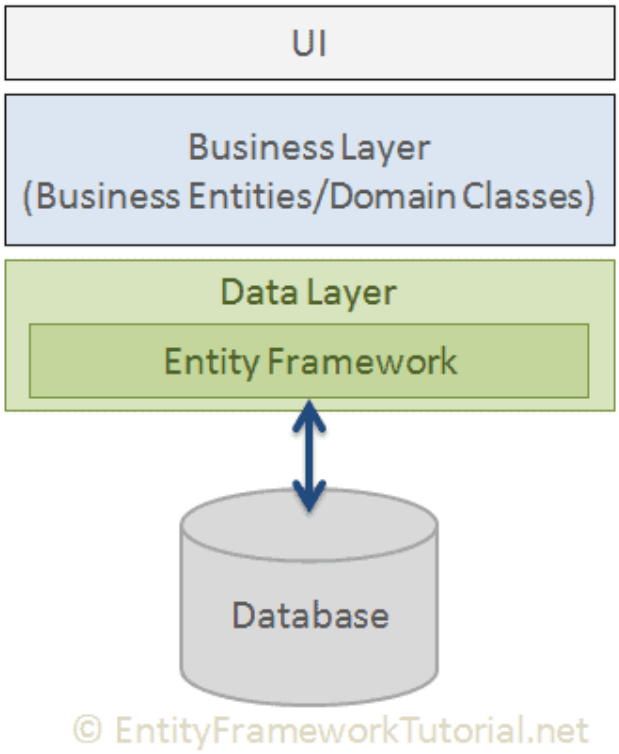

Figura 2.9: Posizionamento di Entity Framework nella struttura di un'applicazione (fonte entityframeworktutorial.net)

RPC (Remote Procedure Call) tra server e client. Nella maggior parte dei casi, la libreria sfrutta le websockets per gestire le varie connessioni. Per comunicare, SignalR utilizza hub, una pipeline di alto livello che consente a client e server di richiamare i metodi l'uno sull'altro. I protocolli hub predefiniti sono due: il primo è un protocollo di testo basato su JSON, il secondo è un protocollo binario basato su MessagePack, un formato di serializzazione più semplice e veloce del normale JSON, anche se non tutti i browser lo supportano. In GPOne viene sfruttato il primo dei due.

2.3.6 Microservizi di GPOne

Si seguirà lo schema dell'architettura in figura 2.10 per descrivere i microservizi e le loro interazioni. Tutti i servizi backend, esposti e non, sono registrati come servizi windows quando viene effettuata l'installazione. I client frontend vengono invece registrati su IIS.

Il servizio API è il servizio potenzialmente più importante dell'intera applicazione, dato che è l'endpoint della chiamate della parte client frontend e in parte anche del client totem. L'API esposta

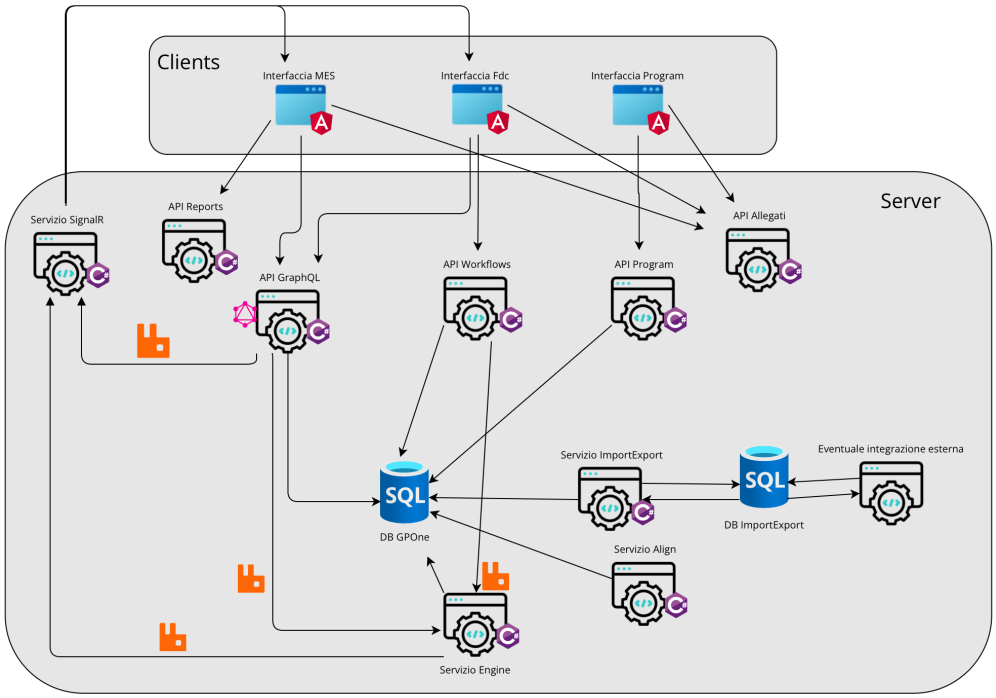

Figura 2.10: Architettura dei servizi GPOne (icone da Icons8 e Flaticon - utente Freepik)

è stata realizzata con GraphQL. Attraverso questo servizio vengono effettuate tutte le operazioni di CRUD. Inoltre all'interno del servizio si sfruttano meccanismi di cache per mantenere lo stato dei permessi degli utenti che hanno effettuato l'accesso al sistema; questi permessi possono essere sia abilitazioni dovute a come il consulente e il cliente hanno voluto gestire i vari utenti, sia le abilitazioni concesse dai moduli attivi nella licenza. L'accesso al sistema viene realizzato mediante un login che sfrutta i token JWT; il token una volta generato viene rimandato al client. Per la parte del client totem, l'api mantiene nella cache anche uno stato delle risorse abilitate su ogni pannello, dove le risorse sono gli operatori e le macchine in officina. La realizzazione dello schema GraphQL viene eseguita con la libreria di automazione Scrutor e con l'utilizzo dei servizi di dependency injection di .NET. Il design pattern alla base dei servizi server dell'applicazione è il command. Nella maggior parte dei servizi di GPOne quando viene fatta una richiesta, viene accodato un command che implementa un'interfaccia generica; una pipeline iniettata in dependency injection si occupa di gestire il command e di richiamare dinamicamente l'handler corrispondente. Le automazioni permettono per le centinaia di entità di registrare automaticamente i command standard di CRUD e le relative mutation e query di GraphQL senza dover creare nuovi file. Qualora

servisse per una query o una mutation un comportamento personalizzato, l'automazione salterebbe la registrazione automatica e andrebbe a prendere l'implementazione effettiva.

Il servizio duale dell'API è il servizio REST. Questo servizio espone una serie limitata di endpoint per la gestione degli allegati. Il record a database dell'allegato viene comunque gestito dal servizio API, ma il trasferimento del file fisico viene gestito dal servizio REST. Questo servizio è diventato obsoleto da un po' dato che GraphQL da un po' di anni permette di gestire anche il trasferimento di file, in futuro è previsto un aggiornamento che prevede di integrare gli allegati sull'API e la rimozione del servizio REST. Il servizio REST è usato da tutti i client, dunque il frontend, il totem e il program hanno mappato nelle loro configurazioni gli endpoint del servizio.

Il servizio ReportAPI espone anch'esso degli endpoint http per la gestione e la stampa di report. Le funzionalità per la gestione dei report sono state implementate con la libreria e l'editor di Stimulsoft. Il servizio PrintingServ viene utilizzato per gestire le comunicazioni con le stampanti dell'ufficio e dell'officina. La comunicazione a questo servizio avviene tramite l'invio di un messaggio usando RabbitMQ, il consumer registrato su PrintingServ dunque riceverà l'evento e si occuperà di mandare in stampa sulla stampante selezionata il documento scelto.

Il servizio ProgramAPI espone l'API REST realizzata per eseguire le operazioni di CRUD dal client di program. Si è deciso di non includere nell'API standard anche questa parte dato che l'API è un servizio già abbastanza complesso e program può anche essere venduto standalone.

Il servizio WorkflowAPI viene utilizzato per la gestione delle procedure dei pannelli totem. Una procedura è una serie di passi che vengono proposti all'operatore a bordo macchina che permettono di dichiarare tutte le informazioni necessarie a registrare le attività svolte nella giornata. La realizzazione è stata fatta esponendo anche in questo caso una API REST, la gestione server delle procedure è stata realizzata tramite la libreria WorkflowCore. WorkflowAPI non arriva tuttavia alla scrittura delle informazioni a database. Il compito in questo caso viene gestito dal servizio engine.

Il servizio engine è il servizio che si occupa di gestire tutte quelle procedure asincrone che possono richiedere tempo, un esempio può essere una ripianificazione delle commesse sul piano di produzione oppure una o più registrazioni provenienti da WorkflowAPI. Queste procedure possono essere lanciate da una mutation del servizio API, che in seguito manderà un messaggio Rabbit all'engine che si occuperà di far partire tutto il processo asincrono, permettendo al mittente del messaggio di ritornare una risposta immediata al client e non bloccare l'utente. Il servizio in

questo caso è anche una sorta di sink per le operazioni di insert/update/delete effettuate dagli altri servizi. Quando viene eseguita una qualsiasi di queste operazioni negli altri servizi, è possibile far mandare un messaggio Rabbit contenente l'entità modificata, i campi e il tipo di operazione. Questo comportamento è simile ai trigger di funzionalità serverless che si possono ricreare in cloud, ad esempio su Microsoft Azure utilizzato le Azure Functions. In ultimo, l'engine si occupa di instanziare gli oggetti driver per la comunicazione con le macchine utensili. L'engine può ricevere informazioni relative allo stato della macchina, se sta lavorando o se è in emergenza, per poi comunicare al client totem di modificare la visualizzazione delle informazioni della risorsa macchina, oppure, come avviene con il program, può direttamente trasferire programmi nella memoria del controllo numerico della macchina.

Il servizio Align si occupa in modo specifico dei ricalcoli asincroni che vengono fatti su tutte le tabelle relative agli ordini di produzioni e alle registrazioni delle attività a bordo macchina. La creazione di questo servizio è stato necessario per evitare di sovraccaricare l'engine di lavoro. Tutti i ricalcoli che svolge questo servizio possono essere disattivabili qualora il cliente voglia effettuare queste operazioni a mano.

Il servizio di SignalR fornisce la comunicazione inversa da server a client. Tutti i servizi backend, qualora dovessero inviare una notifica di avvenuta operazione per il frontend, o un cambiamento di stato di una risorsa per i client totem, possono mandare dei messaggi Rabbit alla coda di SignalR; una volta ricevuto, il servizio comunicherà a uno o più dei client registrati il messaggio creato.

In ultimo, il servizio di ImportExport si occupa dell'integrazione tra GPOne e gli applicativi esterni. L'utilizzo di questo servizio prevede quasi sempre anche l'implementazione di un connettore, che può essere realizzato con un servizio periodico ma può anche essere un job periodico realizzato su SQL (. ImportExport periodicamente inserirà o aggiornerà i record su GPOne in base alle nuove righe che sono state aggiunte nel suo database.

I database utilizzati dai servizi di GPOne sono due, entrambi SQL Server; i database in questo caso vengono installati da cliente nella maggior parte delle volte utilizzando la versione Express, ovvero quella gratuita. Questo ovviamente porta con se tutta una serie di limiti tra cui:

- dimensione massima del DB, un singolo database non può superare i 10 GB;
- bassa parallelizzazione, dato che supporta al massimo 1 socket/4 core, rispetto alla versione

standard che può arrivare fino a 4 socket/24 core;

- nessuna automazione, quindi eventuali job periodici di SQL non possono essere usati.

Chiaramente l'installazione di una versione standard o enterprise di SQL Server è a carico del cliente, qualora ci fossero necessità di rispettare alcuni vincoli di disponibilità e di prestazioni, ma essendo una licenza dal costo importante, in assenza di altre indicazioni si opta sempre per quella gratuita.

Il primo database è quello di GPOne, e contiene oltre un centinaio di tabelle con oltre 10 schemi diversi. La scelta di utilizzare un unico database potrebbe far storcere il naso, specialmente data l'architettura a microservizi scelta. Tuttavia i servizi di GPOne scrivono quasi sempre su tabelle con namespace diversi, quindi nonostante condividano lo stesso database non si sovrappongono con le operazioni di lettura/scrittura evitando in questo modo le race conditions. La scelta di utilizzare un singolo database è stata presa per evitare di dover gestire subito nelle prime iterazioni istanze multiple di database. Nulla vieta di separare i namespaces in database diversi qualora fosse necessario per motivi di spazio o di prestazioni.

Il secondo database è quello di ImportExport. Il database in questo caso conta poco più di 10 tabelle al momento. Sono presenti delle tabelle che forniscono un tracciato standard di inserimento per le informazioni che devono essere poi trasferite su GPOne o al gestionale esterno, oltre che delle tabelle per loggare eventuali esiti positivi e negativi dei processi di importazione ed esportazione dati.

2.4 Architettura e struttura applicazione licenze

Per le funzionalità e i requisiti richiesti, l'applicazione che si deve sviluppare non dev'essere estremamente complessa. L'applicazione delle licenze dev'essere gestita in modo da essere autonoma in questo primo rilascio, ma deve poter evolversi per essere inserita in un contesto distribuito qualora si decidesse di spostare tutta la gestione interna dell'azienda su nuovi software e tecnologie.

Per questi motivi l'architettura che è stata scelta è la Clean. Essendo piccola, è fondamentale che l'applicazione sia scritta in modo molto ordinato, con una struttura che possa anche risultare rigida, ma che sia comprensibile. Seppure un po' più complessa di un monolite tradizionale, le

competenze per gestire questo tipo di architettura ci sono nel reparto Ricerca e Sviluppo. Il numero di sviluppatori assegnato a questa applicazione nelle fasi iniziali è di soli due, è dunque possibile essere più rigidi nello sviluppo dato che le persone da controllare sono poche.

La struttura della soluzione è stata impostata su un unico repository comprensivo di backend e frontend. La scelta in questo caso è stata dettata dal fatto di voler utilizzare un'unica pipeline di compilazione e un'unica pipeline di rilascio, siccome ogni utilizzo delle pipeline ha un costo su Azure Devops. Su GPOne è stato necessario scomporre in più repositories dato che una compilazione di un singolo repository poteva impiegare dai 10 ai 25 minuti, specialmente riguardo la parte backend; nel caso una modifica avesse impattato solamente il frontend, con un'unica pipeline l'attesa per la sola compilazione di tutti i repositories sarebbe stata troppo lunga, specialmente in un team di 10 sviluppatori che fanno una media di una pull request a testa al giorno. Questo problema chiaramente non si presenta su un'applicazione ridotta come quella delle licenze.

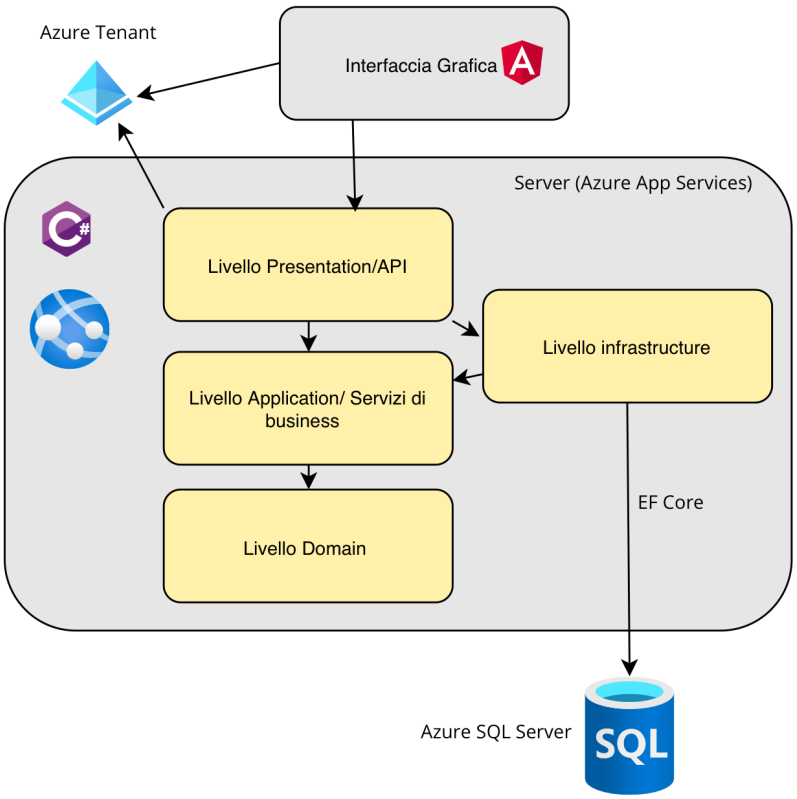

Figura 2.11: Struttura applicazione licenze

Per le tecnologie utilizzate si è cercato di condividere il più possibile lo stack tecnologico di

GPOne. Nel backend è stato utilizzato .NET 9, ma anche in questo caso si aggiornerà nell'agosto 2026 a .NET 10. Le librerie per la gestione dei database e dei log sono le stesse utilizzate su GPOne, quindi EF Core e Serilog. Nel frontend si è utilizzata anche in questo caso la stessa versione di Angular di GPOne, la v20.

La struttura backend è la struttura standard di un'architettura Clean. È presente il livello domain più interno, in cui sono registrate tutte le classi degli oggetti di business e le interfacce di base. Queste classi sono quelle che verranno utilizzate da EF Core per la realizzazione dello schema di database. Nel livello application sono state implementate tutte le interfacce che verranno utilizzate sia dai servizi di business all'interno dello stesso livello che dalle classi del livello infrastructure. Nell'infrastructure è stata implementata tutta la parte di accesso e di gestione delle operazioni di CRUD del database, tutto utilizzando EF Core. Infine il livello presentation contiene tutta la parte di dependency injection iniziale e i controller degli endpoint REST che vengono esposti, oltre alle configurazioni di base dei servizi di log. Siccome il numero di entità previste per l'applicazione è abbastanza ridotto, si è deciso di gestire le operazioni delle singole entità implementando il design pattern Repository. Questo ha standardizzato la gestione delle operazioni delle entità e ne ha separato in modo chiaro le gestioni permettendo così a ogni classe di essere single responsibility.

La struttura frontend rimane pressochè identica a GPOne. La differenza sta in alcune scelte implementative e di stile. Siccome lo sviluppo di GPOne è partito nel 2019, alcune feature delle versioni più avanzate di Angular ancora non erano state sviluppate. In GPOne nella maggior parte del codice la gestione dei reactive form e delle chiamate asincrone sfruttano ancora metodi che Angular stesso ha deprecato da un paio di anni (ad esempio l'utilizzo delle promises). Per l'applicazione licenze la direzione è stata quella di utilizzare la tecnologia dei signals da subito. Inoltre, non essendo un'applicazione visibile ai clienti ma solo ad uso interno, lo stile grafico non segue quello della suite di GPOne o di GP9Over. Il pacchetto grafico che è stato scelto è il pacchetto Sakai sviluppato da PrimeNG. Questo pacchetto fornisce tutta una serie di componenti pronti all'uso per le griglie e i form.

2.5 Ecosistema Microsoft Azure

Come detto in precedenza, OSL ha da sempre sviluppato applicazioni per rilasci on-premises e anche con i software uso interno non si è mai affidata a gestori cloud, ma ha sempre utilizzato server interni. Questo genere di riluttanza allo sfruttare provider esterni di infrastrutture cloud ha portato a una mancanza di competenze non solo nello sviluppo sui servizi cloud, ma anche nella gestione e nella scelta corretta dei servizi. Come primo passo in questa direzione, come Ricerca e Sviluppo si è cercato il provider più adeguato alle esigenze e all'azienda. Inoltre, azienda ha ritenuto utile permettere ad alcuni membri della Ricerca e Sviluppo di studiare e sottoporsi all'esame per ottenere la certificazione AZ-900 Microsoft Azure Fundamentals. La AZ-900 è una certificazione di livello base che introduce, oltre che ai concetti fondamentali del cloud computing, tutti i servizi di Azure più utilizzati, la gestione degli storage di Azure, delle zone di disponibilità e dei servizi di calcolo.

2.5.1 La scelta di Azure rispetto ad AWS

La direzione dell'azienda nel corso degli anni e le tecnologie utilizzate nello sviluppo software hanno indirizzato pesantemente la scelta verso Microsoft Azure piuttosto che su AWS di Amazon. Ovviamente la scelta non è stata orientata dalle prestazioni o da vantaggi tecnologici che uno dei due provider poteva avere sull'altro. Per il carico di lavoro e l'obiettivo che ci si è posto entrambi i providers sarebbero stati capaci di fornire servizi ben più che adeguati. La differenza l'ha fatta l'integrazione delle tecnologie sfruttate in azienda e i costi delle infrastrutture. Come si è spiegato nei capitoli precedenti, i prodotti di OSL sfruttano molte tecnologie di Microsoft, da tutto lo stack .NET a SQL Server; anche in alcuni compiti di gestione interna vengono usati altri prodotti, come PowerBI, la suite Office365, Sharepoint e Azure Devops. La fetta di mercato di Azure è data principalmente da tutte quelle aziende che hanno sfruttato per anni i prodotti Microsoft, ed Azure aggredisce questa parte del mercato fornendo prezzi più competitivi rispetto ad altri provider per spostare le tecnologie nel loro cloud, oltre al vantaggio di mantenere sostanzialmente invariati i flussi di lavoro aziendali. Inoltre Azure permette una gestione molto semplice anche delle implementazioni ibride su stack Microsoft che possono avere le aziende tramite i loro prodotti. AWS è un'ottima scelta qualora l'azienda sia relativamente giovane, poichè è di fatto il leader di mercato dei servizi cloud e anche il più avanzato, ma su aziende con un retaggio storico di servizi

Microsoft servirebbe più tempo per adattarsi alla transizione. Si prenda come esempio un'istanza SQL Server: se l'istanza viene creata sulle infrastrutture Azure, oltre ad avere prestazioni migliori, dato che è SQL Server è una tecnologia proprietaria di Microsoft, il costo si riduce rispetto a AWS addirittura di oltre il 90%, a parità di risorse instanziate. Inoltre, qualora si decidesse di trasferire alcune licenze da server on-premises ad Azure, come licenze SQL Server o Windows Server, si otterrebbe anche un significativo sconto sulla propria sottoscrizione. Chiaramente rimanendo sullo stack Microsoft, il rischio è quello di ritrovarsi in un vendor "lock-in", ovvero nell'incapacità di staccarsi dai servizi Microsoft qualora si decidesse di cambiare provider, poichè troppo propagato all'interno dei flussi aziendali. Per quanto di fatto a livello di gestione interna sia già così, a livello dei prodotti OSL questo problema non dovrebbe presentarsi, dato che sia GPOne che l'applicazione licenze sono stati realizzati in modo da non dipendere da una tecnologia presente solo su Azure.

2.5.2 Azure Devops e Microsoft Entra ID

L'azienda non è nuova al mondo Azure. Seppure non avesse mai sviluppato applicazioni da rilasciare in cloud su infrastrutture Microsoft, da anni sfrutta alcuni dei suoi servizi. Non ci si soffermerà sulla suite Office365 o su Sharepoint, che comunque hanno un peso non indifferente nei flussi di lavoro di OSL. Ciò che accomuna però questi servizi è l'utilizzo di Microsoft Entra ID, precedentemente chiamato Azure AD (Active Directory). Entra ID è il servizio di gestione degli accessi e delle identità di Microsoft. Siccome le mail aziendali sono gestite tramite Outlook di Microsoft, Entra ID permette di unificare il login implementando il SSO (Single Sign-On) e il MFA (Multi-Factor Authentication) per tutti i prodotti Microsoft utilizzati in azienda. All'interno di Azure è stato creato un tenant per OSL; in questo modo vengono anche gestiti eventuali permessi per le applicazioni. La gestione di Entra ID all'interno dell'azienda viene effettuata dal reparto IT di Overmach.

A livello di analisi e sviluppo software, l'azienda si è legata ad Azure Devops da quando è iniziato lo sviluppo di GPOne, nel 2019; nell'ultimo anno, la gestione di altri prodotti come GP9Over è stata anch'essa spostata su Devops. Azure Devops è una suite che permette di gestire il ciclo di vita di un prodotto software. Il prodotto mette a disposizione una serie di funzionalità per facilitare la supervisione e la realizzazione degli sviluppi:

- boards, dove vengono inseriti i vari tasks e le varie feature da realizzare dagli analisti, la

gestione avviene attraverso varie dashboard stile kanban;

- repos, dove vengono gestiti i vari repositories degli sviluppi che Azure integra con Git;
- pipelines, dove si gestiscono tutte le pipelines di CI/CD (Continuous Integration/Continuous Delivery)
- artifacts, dove vengono salvati gli artifacts (pacchetti o librerie software) che vengono prodotto dagli sviluppatori del team;
- test plans, dove è possibile gestire tutti i test da effettuare sui software gestiti in devops.

Le pipeline di Devops vengono gestite tramite degli agents (i processi di Azure che gestiscono i vari step di pipeline) installati sulle macchine virtuali dei server interni di OSL. All'interno del progetto di Devops su cui sono gestiti GPOne e l'applicazione licenze, possono essere eseguite due pipeline in parallelo. Un'eventuale aggiunta di esecuzione delle pipeline in parallelo necessita di un pagamento aggiuntivo. All'interno della sezione repos di Devops è possibile modificare le politiche per i singoli repositories o addirittura per un sottoinsieme di branches di un dato repository; tra queste, è possibile gestire le modalità di pull requests oppure abilitare solo alcuni metodi di commit delle modifiche su Git.

2.5.3 Subscription e risorse di Azure

Per iniziare a usare i servizi di Azure, è necessario avere una Azure Subscription, o sottoscrizione. Nell'ecosistema di Azure i servizi sono gestiti come una struttura gerarchica per ogni cliente in cima alla quale è presente una o più sottoscrizioni. A ogni sottoscrizione viene associato un metodo di pagamento che dev'essere collegato a un conto corrente. È possibile avere conti diversi su sottoscrizioni diverse. Questo è particolarmente utile qualora si volessero dividere in base ai reparti o ai gruppi di lavoro i metodi di pagamento, in modo che ogni reparto può attingere solo da un unico conto. Per monitorare diverse sottoscrizioni sotto un unico blocco è possibile creare a un livello superiore un gruppo di gestione (Azure Management Group), in modo da visualizzare i dati dei consumi aggregati delle sottoscrizioni; il gruppo di gestione non è stato necessario ai fini della gestione licenze.

Scendendo nella gerarchia, si incontrano i gruppi di risorse (Azure Resource Groups). Questi, analogamente ai gruppi di gestione, fungono da contenitori logici per raggruppare le risorse. Infine le risorse sono l'unità di lavoro di Azure. Una risorsa è una qualsiasi entità o servizio che viene creato all'interno di un gruppo di risorse di Azure, come può essere una macchina virtuale, un gestore di containers o un database. È obbligatorio che una risorsa sia inserita in un gruppo di risorse. Inoltre è necessario per ogni risorsa indicare in quale zona di disponibilità deve esistere.

Una zona di disponibilità (Azure Availability Zone) è una struttura fisica su cui vengono gestiti i vari servizi istanziati, quindi all'atto pratico è un data center. Questa è inserita in una regione (Azure Region). Il concetto di zona di disponibilità è fondamentale per garantire che un'applicazione possa continuare a operare anche in caso di eventi imprevisti. Ogni zona è isolata e a diversi chilometri di distanza da un'altra zona nella stessa regione. Solitamente Azure offre diversi modi per fornire la business continuity di un'applicazione, la più semplice è la replicazione di un servizio in più zone all'interno della stessa regione. Inoltre, per applicazioni particolarmente critiche, le linee guida di Azure suggeriscono addirittura di replicare i servizi in regioni diverse.

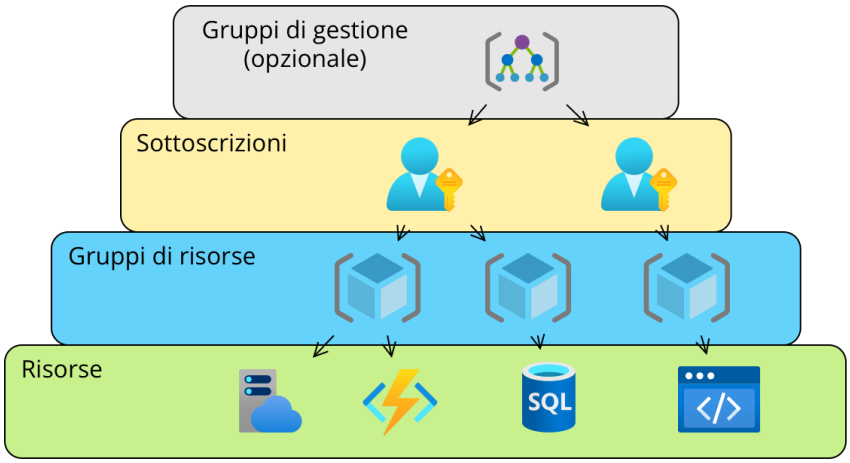

Figura 2.12: Gerarchia Microsoft Azure

Per la gestione dell'applicazione licenze, non era necessario istanziare una struttura in Azure che fosse troppo complicata. È stata creata una singola sottoscrizione per le risorse necessarie all'applicazione. Si è deciso di utilizzare due gruppi di risorse: il primo è utilizzato come ambiente di staging e test, quindi adibito all'utilizzo da parte della Ricerca e Sviluppo e del reparto test; il secondo contiene tutte le risorse per il rilascio in produzione.

2.5.4 Servizi Azure per applicazione licenze

Si introdurranno ora tutte le risorse che sono state utilizzate all'interno dei due gruppi di risorse.

Il primo servizio utilizzato è Azure App Services. App Services è un servizio PaaS (Platform-as-a-Service) che permette allo sviluppatore di eseguire direttamente un'applicazione fornendo semplicemente il codice della stessa. Ciò ha il vantaggio di demandare tutta la gestione dell'infrastruttura sottostante ad Azure stesso, dunque eventuali aggiornamenti di macchine virtuali o patch di sicurezza vengono gestiti direttamente dal provider. In fase di creazione della risorsa App Services è possibile personalizzare una serie di parametri per l'infrastruttura tra cui:

- i core di vCPU, il numero di processori virtuali;
- memoria RAM e di archiviazione;
- il numero di istanze massime che possono esistere contemporaneamente;
- il sistema operativo, in questo caso Windows Server o Linux;
- la percentuale di disponibilità annuale, ovvero i vari livelli di SLA possibili;
- la regione geografica su cui verranno gestite le istanze.

La scelta del numero di istanze, della disponibilità e della regione geografica sono fondamentali per poter rispettare eventuali requisiti di prestazioni o di legge. Un numero di istanze maggiori o una disponibilità maggiore permette di garantire più continuità di servizio e una risposta migliore all'aumentare del numero di utenti. La scelta della regione dev'essere fatta non solo in ottica di vicinanza agli utenti (nel caso di OSL gli eventuali utenti si collegherebbero principalmente dall'Italia dunque è fondamentale utilizzare un data center in Italia o quantomeno in un paese vicino), ma serve anche qualora per legge per alcuni servizi sia necessario che i dati risiedano in un determinato paese. Tramite le impostazioni di App Services è possibile impostare in automatico eventuali criteri di scalabilità dell'applicazione che si attivino all'aumentare della richiesta o qualora un data center non fosse più disponibile.

All'effettivo, Azure App Services si divide in due sottorisorse: la prima è App Services Plan e può essere considerato un contenitore in cui sono definiti tutti i criteri dell'infrastruttura; la seconda

è l'App Service effettiva, ovvero l'applicazione che viene eseguita. Questa struttura permette con un solo App Services di poter eseguire più applicazione sfruttando la stessa risorsa di calcolo sottostante. Questo può essere utile per avere due versioni della stessa applicazione, una di test e una di produzione. Per l'applicazione licenze si è preferito tenere due App Services separati in gruppi di risorse diversi, allo scopo di monitorare separatamente i gruppi di risorse. In entrambi gli App Services, il sistema operativo scelto è Linux, la distribuzione è gestita da Azure. Sull'App Services di staging la versione utilizzata è la Basic B1, mentre per quello di produzione si è optato per la versione successiva Basic B2. Queste versioni sono utilizzate da Azure per identificare categorie di infrastrutture che forniscono la medesima capacità di calcolo, in tabella **??** è possibile osservarne le differenza; chiaramente si è optato per più prestazioni nella versione di produzione, ma per il carico previsto non è stato necessario andare su versioni più costose.

Risorsa	**Basic B1**	**Basic B2**
vCPU	1	2
RAM	1,75 GB	3,5 GB
Storage	10 GB	10 GB
Max istanze	3	3
Disponibilità %	99,95 %	99,95 %
Costo orario	0,018 USD	0,036 USD

Tabella 2.1: Differenze tra versioni Basic B1 e B2 di Azure App Services (Gennaio 2026)

Un argomento che è stato affrontato in sede di scelta delle tecnologie è stato se sviluppare o meno l'applicazione con i containers. Un container è un pacchetto software che contiene tutto il necessario per eseguire un'applicazione in qualsiasi ambiente. Nel container vengono definite le dipendenze di un'applicazione e il suo codice; una volta creato il container, è possibile rilasciarlo in modo molto veloce dove necessario. Questo approccio si porta dietro una serie di benefici molto importanti, come la portabilità, l'isolamento dei servizi e la leggerezza di ogni container. Se si prende come esempio GPOne, è possibile rendere ogni microservizio un container. Tuttavia in OSL questo approccio non è stato perseguibile per due motivi. Il primo e il più importante è la mancanza

di competenze per gestire questo modo di sviluppare e rilasciare le applicazioni. Bisogna tener conto che gli sviluppatori in azienda non hanno mai sviluppato diversamente dall'eseguibile compilato e installato su Windows. Il secondo è che sarebbe un approccio sovraingegnerizzato per lo stato attuale delle applicazioni vendute; installare un orchestratore sui server clienti non porterebbe al momento alcun beneficio, specialmente se l'assistenza verrebbe effettuata da consulenti e reparto assistenza. Dunque si è preferito non prendere in considerazione Azure Container Apps o sfruttare App Services con i containers. Chiaramente sia per GPOne che per l'applicazione licenze non ci si è preclusa la possibilità di convertirsi ai containers in futuro.

Il secondo servizio utilizzato è Azure SQL Server. Il nome del servizio è abbastanza autoesplicativo. Come nel caso di App Services, anche in questo caso si tratta di un PaaS, in cui viene fornito il motore di database di SQL Server. Il servizio viene fornito con una licenza SQL Server Standard. Nel motore è possibile gestire più database, fintanto che non si supera la quota di memoria fornita dal livello di servizio scelto. Parlando dei livelli, in questo caso vengono messi a disposizione due modelli di acquisto:

- modello basato sui vCore, ed è simile alla scelta di cpu, ram e memoria che si aveva in App Services;
- modello basato sui DTU (Database Transaction Unit), in questo caso viene messa a disposizione una misura di calcolo combinata di cpu, memoria e operazioni di lettura e scrittura, per ricondursi a carichi di lavoro tipici, per tutte quelle applicazioni che rientrano in un caso abbastanza noto di gestione o per i clienti che preferiscono una serie di scelte preconfigurate.

Per entrambi i motori di database di staging e produzione, si è optato per un modello basato su DTU e con livello di servizio Basic, che Azure identifica per i carichi di lavoro meno complessi. La dimensione totale dei database prevista dal livello Basic è di 2 GB.

Sia per Azure App Services che per Azure SQL Server, la regione in cui sono gestiti i servizi è Italia Nord.

Il terzo servizio utilizzato è Azure Static Web App. Questo servizio è molto semplice dato che similmente a quanto succede per App Services, serve ad eseguire applicazioni, in particolare questo servizio esegue pagine web statiche. In questo caso il servizio non viene rilasciato solo in una regione ma viene gestito in tutte le regioni direttamente. Il servizio permette poi di interfacciarsi

con delle API rilasciate nella stessa Static Web App, oppure in altri servizi nella rete di Azure come App Services. Per l'applicazione licenze quindi il servizio gestisce l'interfaccia frontend e la comunicazione con l'API backend rilasciata su App Services.

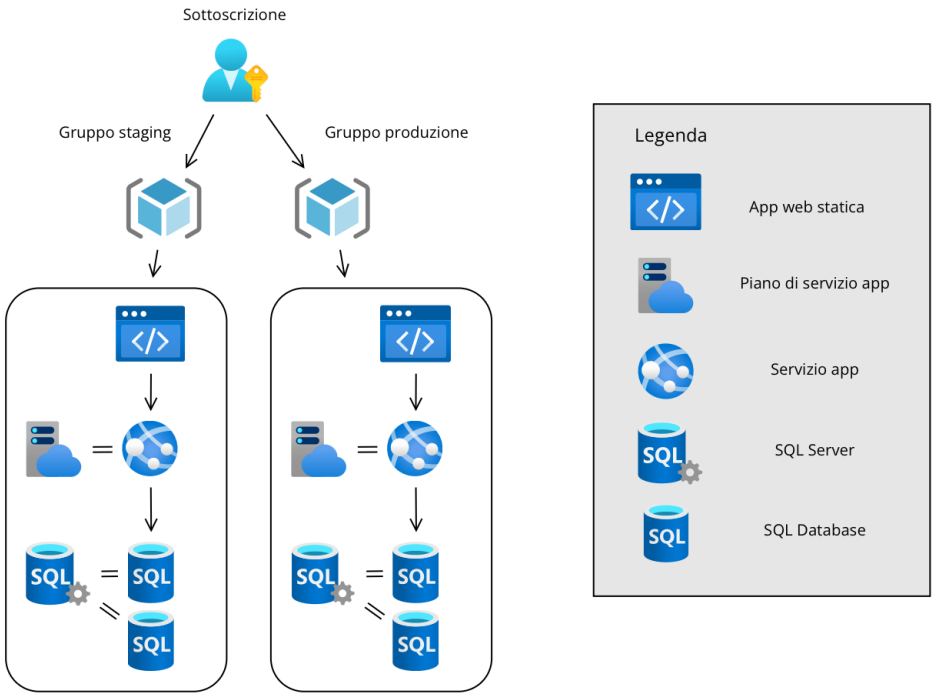

Figura 2.13: Struttura in Azure dei servizi per gli ambienti dell'applicazione licenze

Da figura 2.13 schema azure si può quindi osservare l'insieme delle risorse di Azure e come comunicano all'interno della rete di Azure.

Un ultimo cenno va fatto ad altri due servizi che sono stati utilizzati, seppure non istanziati separatamente ma inclusi nei servizi già citati in precedenza. Azure Application Insights fornisce il monitoraggio dei vari servizi. Attraverso questo servizio è possibile raccogliere i dati di telemetria generati dall'applicazione e generare alert in automatico qualora si verificassero eventi inattesi come aumenti dei tempi di risposta oppure visualizzare gli eventi dell'applicazione in base a un orizzonte temporale. Azure Budgets è un servizio che permette di tenere sotto controllo le spese delle risorse. Questo servizio rientra nella categoria di servizi di Azure Cost Management. Questo strumento permette di verificare il costo effettivo delle risorse nel mese e il costo previsto a fine mese, oltre a impostare notifiche qualora si superi una determinata soglia.

Nel monitoraggio dell'applicazione licenze viene sfruttato spesso in staging il log delle chiamate rest che hanno generato degli errori 500, mentre in fase di sviluppo soprattutto è stato impostato un alert che si occupava di notificare tramite mail quando i costi di Azure App Services e di SQL Server superavano i 15 USD mensili.

2.5.5 Calcolo costi e preventivi

Nelle scelta di quali servizi utilizzare è stato usato Azure Cost Calculator. Questi altro non è che un preventivatore dei costi di utilizzo dei servizi di Azure disponibile per gli utenti. Con questo calcolatore è possibile per ogni servizio impostare la quantità di risorse di calcolo desiderata e osservare le variazioni tra le varie combinazioni.

Lo strumento è stato fondamentale sia nella scelta della configurazione di App Services che di SQL Server. Per Azure App Services, la scelta del sistema operativo è stata quella di maggiore impatto sui costi. Scegliendo la versione Basic B1, se si fosse optato per un sistema operativo Windows il costo orario per istanza sarebbe stato 0,075 USD, che come costo mensile (730 ore) avrebbe portato a 54,75 USD, moltiplicato per massimo 3 istanze. Scegliendo Linux invece, il costo orario per istanza si è ridotto drasticamente a 0,018 USD, con un costo mensile di 13,14 USD, quasi 4 volte in meno rispetto alla versione con Windows.

Per Azure SQL Server l'impatto sul costo è stato dato da due impostazioni: modello di costo e livello di calcolo. Il modello di costo scelto ha impattato significativamente sia sul costo che sulla complessità di configurazione del servizio. Con il modello vCore con livello di calcolo provisioning, dunque disponibile sempre, e impostazioni minime di vCore e ridondanza e hardware preventivava come media una spesa di circa 390 USD mensili. Passando ovviamente al livello di calcolo ad ambienti senza server il prezzo si riduceva enormemente anche sotto i 2 USD mensili. Tuttavia sfruttando il modello di costo in base ai DTU si semplificava enormemente la scelta delle impostazioni; con il livello di servizio Basic il costo preventivato medio mensile è stato sotto i 10 USD, anche aggiungendo svariati mesi di retenzione di eventuali backups dei database.

Per Azure Static Web App invece, grazie al calcolatore di costo si è visto che il livello base era gratuito. Questo ha quindi pilotato la scelta sull'esecuzione del frontend su questo servizio, in modo da non dover eseguire anche l'interfaccia su App Services e ha permesso di prendere una versione meno potente e più economica di quest'ultimo

2.6 Struttura file licenza e protezione

Si è accennato nella parte dei requisiti a come l'effettivo file utilizzato per controllare la sia fondamentale per risolvere alcune delle criticità del flusso di rilascio e di aggiornamento clienti.

La ristrutturazione del file di licenza passa attraverso due punti fondamentali: la struttura in se, quindi i dati contenuti all'interno della licenza, e come vengono nascoste, o quantomeno come vengono protette da eventuali manomissioni.

Abbiamo già spiegato la necessità di normalizzare la struttura di gestione dei moduli a un unico livello. Quindi la struttura vista nel primo capitolo in figura 1.2 deve essere appiattita. Non ci saranno più applicazioni e moduli dell'applicazione ma solo moduli. Tuttavia vanno comunque distinti due categorie di moduli: quelli che pilotano un numero di abilitazioni e quelli che sono semplicemente di abilitazione di funzionalità sull'applicazione. Dunque un eventuale entità di modulo deve contenere le informazioni del codice del modulo, la data di scadenza per singolo modulo, il numero di abilitazioni fornite da quel modulo se previsto.

La licenza deve fornire anche una serie di informazioni sull'installazione prevista del cliente. Quindi si è deciso che il file di licenza deve contenere:

- il codice cliente per identificare l'installazione cliente;
- la versione della licenza, corrispondente alla versione che verrà installata di GPOne;
- una data di scadenza per tutta la licenza, qualora si volesse gestire un'unica data di scadenza e non singole date per i moduli;
- un flag a uso esclusivo di sviluppatori e test che permette di eludere il controllo della versione in modo che una licenza possa essere sfruttata per più versioni in fase di test;
- la lista dei moduli abilitati, con le relative informazioni già elencate in precedenza.

Il secondo punto riguarda la modifica al metodo per crittografare i dati in licenza. Il precedente metodo non permetteva flessibilità in caso di migrazione dell'installazione di GPOne su macchine virtuali diverse, inoltre qualora servisse fare un cambio al volo del file di licenza necessitava dell'intervento di uno sviluppatore perchè era richiesto un tool che rifacesse l'operazione di rilascio

licenza. Considerando il fatto che si prevede che la maggior parte delle licenze sarà controllata periodicamente da remoto, si è deciso di togliere questa rigidità nel metodo per crittografare la licenza. La strada che si è scelta è una soluzione ispirata al metodo di funzionamento dei token JWT e della crittografia asimmetrica.

Il file di licenza sarà strutturato come un file JSON con due parametri: la licenza criptata e la chiave pubblica. Il processo di ottenimento del dato di licenza criptato è diviso in 2 passi:

- l'applicazione della firma digitale ottenuta tramite la chiave privata applicata al dato di licenza serializzato;
- la codifica dell'insieme di licenza serializzata e firma digitale.

Il processo di decodifica e controllo di validità si otterrà nel processo inverso:

- viene applicata la decodifica al parametro nel file di licenza;
- ottenuta la decodifica viene controllata dall'applicazione tramite la chiave privata la firma digitale; se la firma è verificata la licenza sarà deserializzabile e utilizzabile, altrimenti il software non sarà utilizzabile.

Le funzioni per tutto il processo vengono inserite in un pacchetto NuGet .NET che viene condiviso sia dal progetto di GPOne che dall'applicazione licenze. In questo modo le funzioni saranno sempre allineate all'ultima versione su entrambi i progetti. La chiave privata sarà generata e mantenuta all'interno dell'Azure Key Vault, uno storage protetto di Azure per gestire dati e chiavi che non devono risiedere sull'applicazione stessa. L'applicazione nel momento in cui dovrà applicare la firma si collegherà al Key Vault.

2.7 Flussi di inserimento, rilascio e controllo licenza

L'applicazione deve avere permessi diversi in base al gruppo di appartenenza dell'utente all'interno dell'organizzazione. Fortunatamente, i gruppi dei dipendenti di OSL, quindi i reparti, sono già censiti sul tenant in Azure grazie ai vari prodotti Microsoft già usati in azienda. Dunque sfruttando l'accesso tramite Azure per accedere all'applicazione, indirettamente si riesce anche a conoscere il reparto di appartenenza dell'utente.

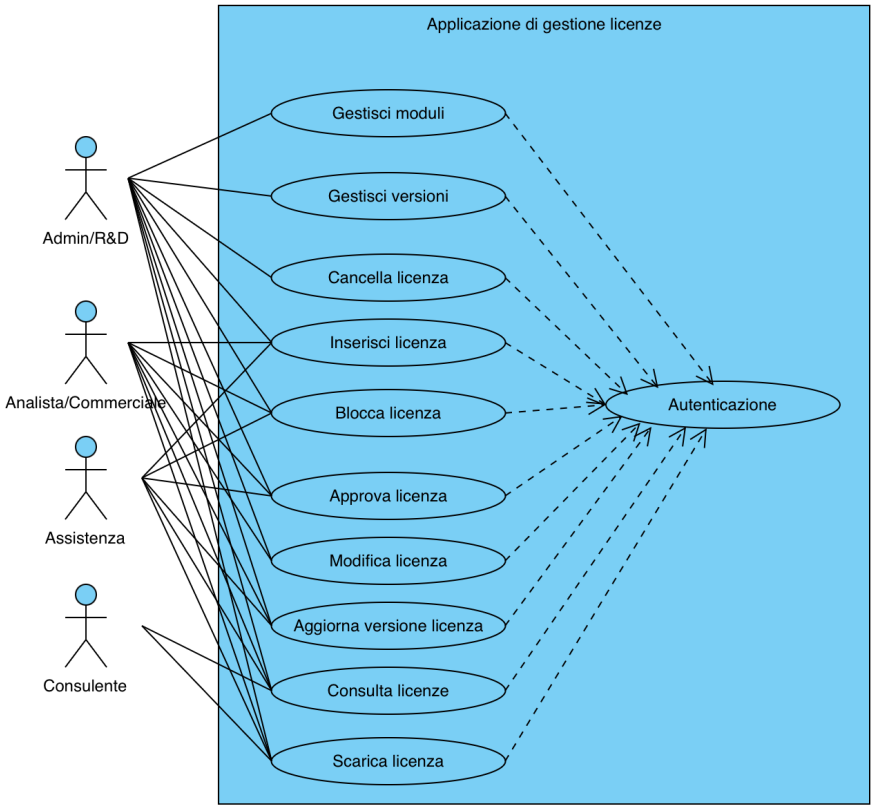

Figura 2.14: Diagramma UML per l'applicazione licenze

Sfruttando figura 2.14 si possono osservare i diversi permessi che devono essere forniti. L'utente admin, che in questo caso corrisponde sicuramente almeno al reparto Ricerca e Sviluppo, deve poter fare tutto. Chiaramente essendo il reparto incaricato alla gestione e allo sviluppo, deve essere possibile sia in staging che in produzione poter usare tutte le funzionalità dell'applicazione. In particolare, questo gruppo di utenti dev'essere l'unico a poter inserire e modificare moduli e versioni, in quanto l'aggiunta di uno dei due deve per forza passare prima da uno sviluppo su GPOne.

Il gruppo che comprende analisti e commerciali devono poter gestire tutto delle licenze. Questo perchè il commerciale può modificare l'offerta al cliente e dunque deve poter gestire l'eventuale aggiunta di un modulo o di un numero di abilitazioni in modo abbastanza veloce. Lo stesso discorso si può fare per gli analisti. Dev'essere anche possibile per queste categorie modificare o aggiornare la licenza e nel caso bloccarla. Inoltre devono poter eseguire l'approvazione della licenza, quindi il passaggio in uno stato che permetta di scaricare la licenza e di utilizzarla.

Il gruppo assistenza ha sostanzialmente diritti analoghi a quelli di commerciali ed analisti, dato che sono il supporto di primo livello sia per i clienti che per i consulenti. Non devono poter però modificare la licenza, dato che quel tipo di azione dev'essere motivata da una decisione commerciale o funzionale.

Il gruppo consulenza deve avere i diritti più essenziali; devono chiaramente poter consultare le licenze, idealmente solo dei clienti a cui sono assegnati, e devono poter scaricare il file di licenza oppure ottenerne la chiave di attivazione, siccome sono le figure delegate all'installazione del prodotto presso il cliente. In questo modo possono essere il più autonomi possibile durante le installazioni.

Analizzati gli use case dell'applicazione delle licenze, è necessario definire tutti i flussi che si dovranno implementare tra le varie applicazioni e che verranno seguiti dai vari reparti.

Il primo e più importante sarà il flusso di inserimento sull'applicazione di una nuova licenza o di una preesistente. Questa richiesta come già detto in precedenza può arrivare dal reparto commerciale a seguito di un ordine cliente. All'atto pratico la richiesta può arrivare da GP9Over e quindi andare a riportare le informazioni sull'applicazione licenze o viceversa.

Nello diagramma in figura 2.15 si prende in esame il caso in cui si inserisca o si modifichi una licenza sulla nuova applicazione. Il passaggio inverso da GP9Over è molto simile ad eccezione dell'ultimo step, quindi lo schema di quel flusso è stato omesso in quanto sostanzialmente uguale a quello per l'applicazione nuova.

Il primo passaggio è quello di censimento del cliente; tendenzialmente le informazioni del cliente sull'applicazione difficilmente vanno aggiornate in quanto viene mantenuto solo un codice interno di riferimento del cliente e una descrizione, dunque è un semplice controllo di esistenza del record.

Se la licenza esiste già, i due controlli che vengono fatti sono sui moduli e sulla versione. Per i moduli è possibile attivarli o disattivarli, o modificare la data di scadenza dei singoli moduli, oltre che della testata della licenza stessa. Questa modifica è stato deciso che non necessita di un'approvazione da parte del reparto commerciale o analista, e che quindi può essere rilasciata direttamente dall'aggiornamento successivo dei dati della licenza da cliente. Per l'aggiornamento versione, così come per la creazione dell'intera licenza, è necessario invece il passaggio intermedio di approvazione. Questo da il tempo di controllare che la versione di installazione sia la più corretta

per il cliente e da il tempo di effettuare modifiche sui moduli, dato che ci possono essere modifiche all'ordine cliente fintanto che non si è prossimi alla data d'installazione.

Una volta approvata la licenza viene generato il codice di attivazione ed è possibile scaricare qualora fosse necessario il file di licenza. Infine vengono riportate le informazioni su GP9Over, che come si è accennato, non sarà eseguito direttamente dall'applicazione licenze ma sarà eseguita una chiamata periodicamente una o più volte al giorno da GP9Over che si leggerà tutte le modifiche effettuate dall'ultima chiamata di controllo.

Il secondo flusso riguarda quello che viene eseguito dal programma di installazione dei GPOne e dal consulente che si occupa dell'installazione. L'installazione può essere comunque effettuata in caso di modifiche particolarmente impattanti da una versione all'altra anche da qualcuno di specializzato nell'assistenza o in casi più complessi anche da uno sviluppatore.

Il flusso in figura 2.16 in questo caso si divide in due strade quasi completamente parallele. Nel caso più semplice in cui il cliente non abbia una rete aperta o non fornisca almeno una porta attraverso il firewall aziendale all'applicazione delle licenze, l'installatore dovrà scaricarsi a priori il file fisico della licenza e dovrà sottoporlo al programma d'installazione; in questo caso, le date di scadenza corrisponderanno esattamente a quelle indicate sull'applicazione licenze. Fatto ciò verranno installati o aggiornati i servizi sul server.

Nel caso opposto in cui il cliente fornisca una comunicazione con l'API delle licenze, al consulente basterà inserire il codice di attivazione nel programma. In questo punto viene richiesto prima di procedere se attivare o meno il microservizio di controllo. Per le prime installazioni si fornisce questa possibilità di disattivarne l'abilitazione in caso di eventuali problemi e di lasciare solo la scorciatoia di inserimento del codice di attivazione; è previsto nel momento in cui l'intero processo sarà stabile di installare sempre il microservizio in caso di installazione con il codice.

Qualora il servizio fosse abilitato, alla fine dell'installazione viene fatta una chiamata all'API delle licenze in cui per il codice di attivazione viene salvato anche un identificativo del server su cui è stato installato. Questo serve a monitorare eventuali spostamenti della licenza; può succedere migrando la macchina virtuale su cui risiede GPOne, e in questo caso viene segnalata una notifica sull'applicazione delle licenze per quel cliente, ma è possibile in quel caso resettare l'id del server associato alla licenza. Il controllo è chiaramente necessario qualora si provi a replicare la licenza in modo illecito su altri server. In tal caso la notifica permetterebbe di contattare il cliente e in casi

estremi di bloccare la licenza. Qualora il servizio fosse attivo, la licenza verrebbe salvata con le date di scadenza diverse rispetto a quelle indicate sull'applicazione remota. Si è scelto di spostare la data di scadenza ogni volta a 7 giorni dall'oggi. Questa scelta è stata fatta poichè possono esserci problemi non previsti al microservizio o all'applicazione remota, che impedirebbero l'aggiornamento della licenza; dunque la licenza in questo caso sopravviverebbe per 7 giorni dall'ultima lettura. Dando il tempo ai servizi di riprendersi e non bloccando il cliente.

L'ultimo flusso riguarda il controllo delle modifiche alla licenza una volta che GPOne diventa operativo. Seguendo figura 2.17 il primo passo si collega direttamente a quanto detto in precedenza, controllando che l'id del server corrisponda a quello memorizzato per quel codice di attivazione.

Da qui si diramano tre possibili strade. La prima è quella in cui la licenza sia ancora in corso di validità; in questo caso viene modificata la data di scadenza a 7 giorni dal giorno della lettura e vengono aggiornate le informazioni sui moduli. La seconda si ha nel caso in cui la testata della licenza sia scaduta o siano scaduti tutti i moduli che la compongono. In questo caso la decisione è di non bloccare completamente il cliente ma di segnalarlo tramite una notifica o un banner su GPOne abbastanza invasivo, poichè magari ci sono stati rallentamenti o problemi nel pagamento oppure i servizi remoti non sono operativi da oltre una settimana. Dopo due settimane dalla scadenza i microservizi si spengono automaticamente.

La terza riguarda il caso in cui la licenza sia stata volutamente bloccata dai commerciali o dall'assistenza. In questo caso la licenza aggiornata viene modificata in modo diverso causando lo spegnimento di tutti i microservizi. È chiaramente il caso più drastico e tendenzialmente riguarda questioni di insolvenza o di "rottura dei rapporti" tra cliente e commerciale.

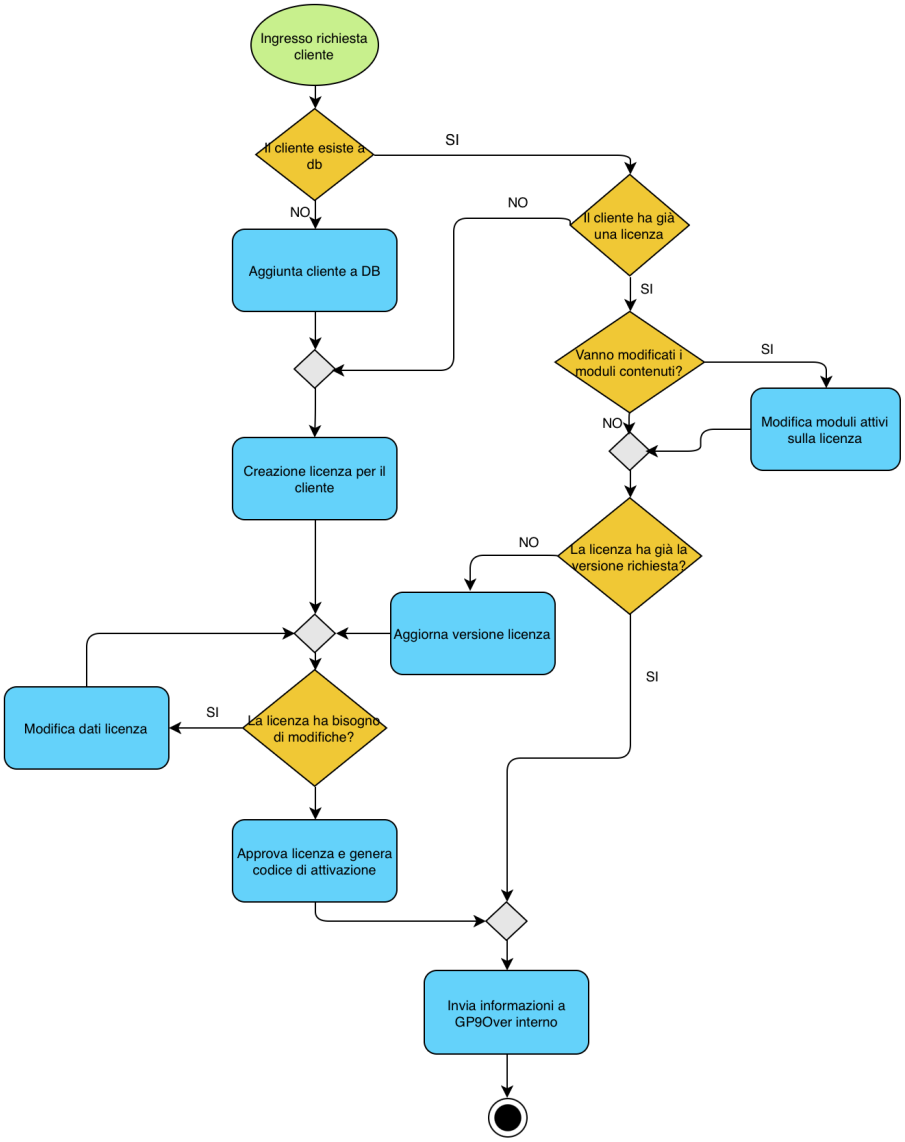

Figura 2.15: Activity diagram per inserimento licenze su nuova applicazione

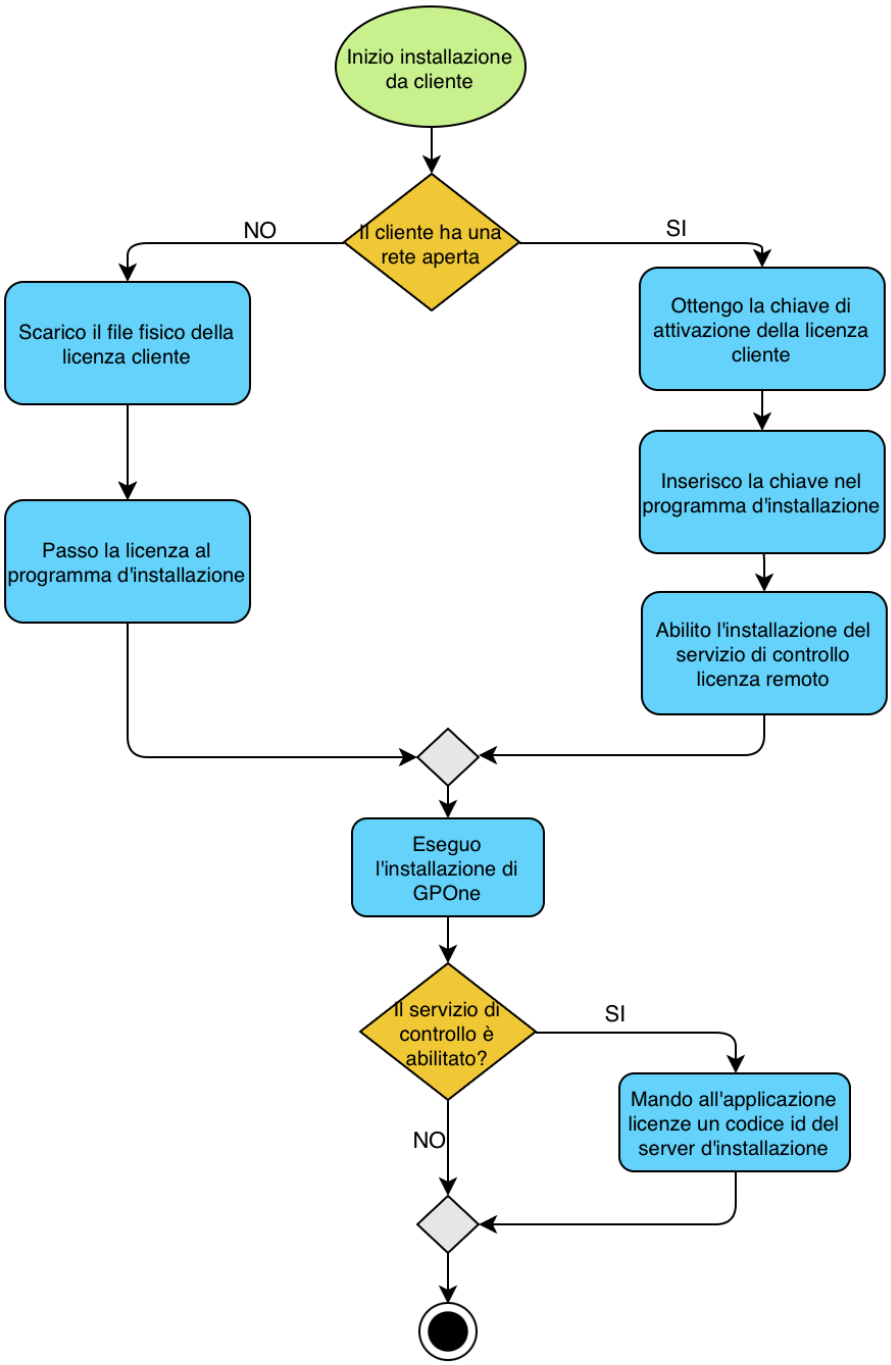

Figura 2.16: Activity diagram per installazione GPOne presso cliente

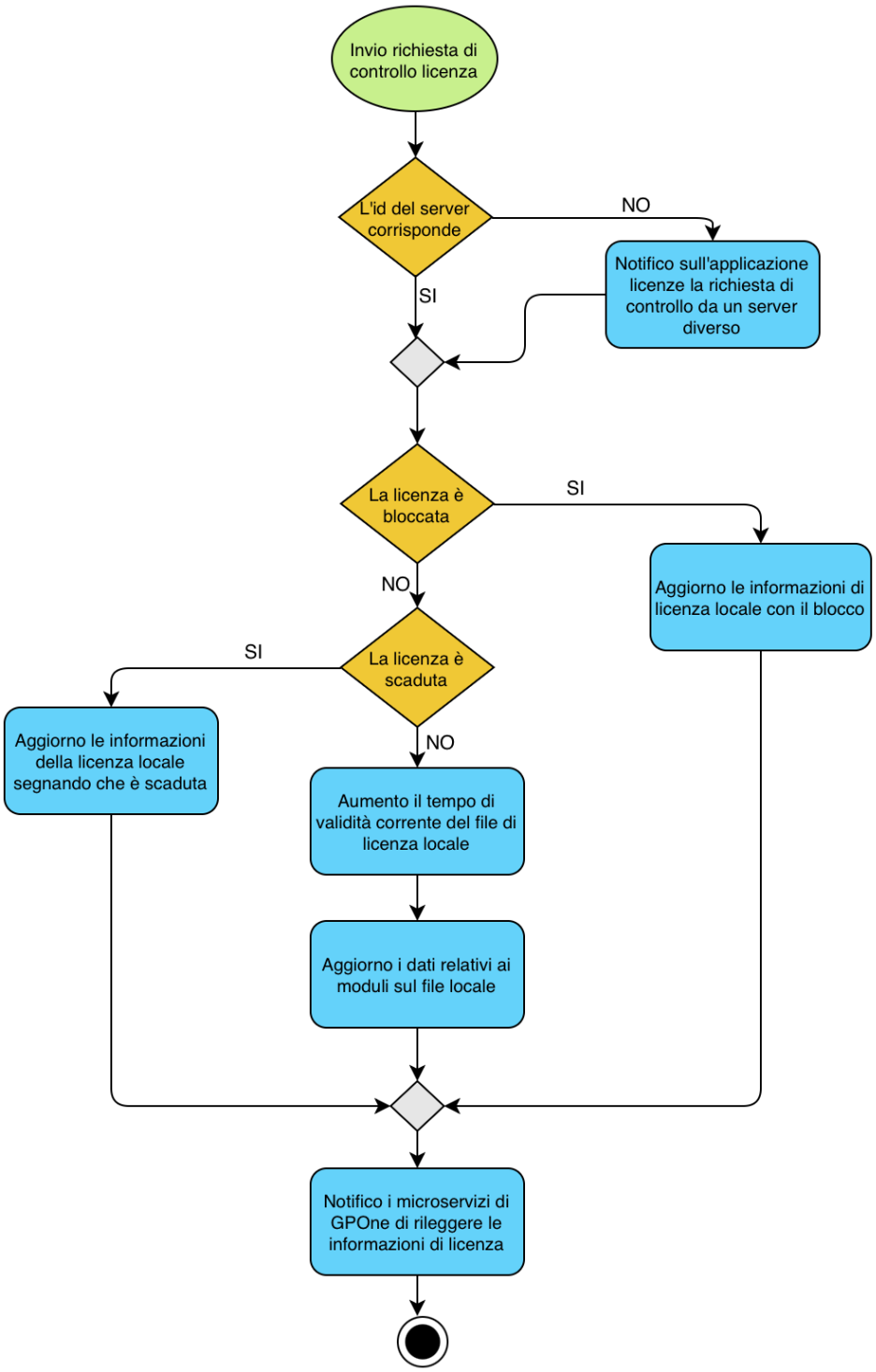

Figura 2.17: Activity diagram per installazione GPOne presso cliente

3. Implementazione dell'applicazione

3.1 Struttura del database

La struttura del database per l'applicazione delle licenze non è particolarmente complesso dato che in questa prima iterazione ha il solo scopo di amministrare le licenze di GPOne.

Le tabelle si divideranno per semplicità in due categorie: tabelle di business e tabelle di audit. Essendo un database molto basilare, le tabelle sono tutte sotto lo schema di default "dbo". Per tutte le tabelle sarà presente una colonna INT Id, ad eccezione delle tabelle molti a molti che chiaramente ne avranno 2.

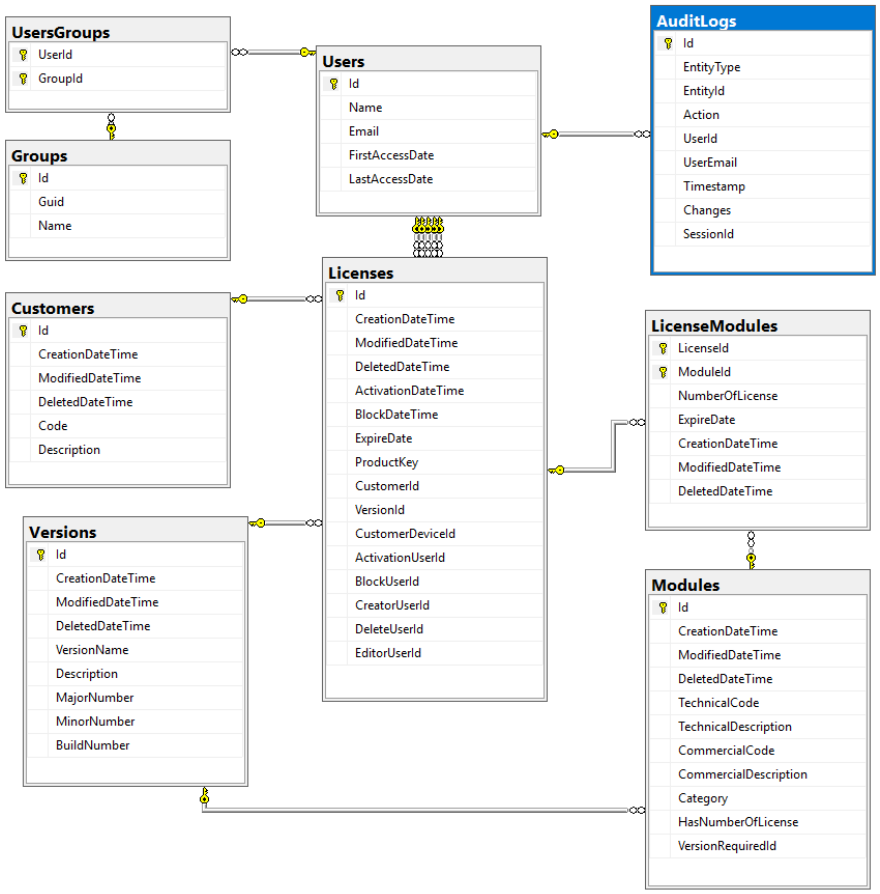

Figura 3.1: Schema tabelle database applicazione licenze

3.1.1 Tabelle di business

Le tabelle di business sono chiaramente le tabelle collegate alla gestione licenze. Le tabelle di questa categoria vengono gestite con il meccanismo della "soft delete", che consiste nel non cancellare fisicamente un record a database, ma valorizzarne una data di cancellazione. Oltre alla data di cancellazione, queste tabelle sono provviste di altre due colonne datetime, data di creazione e data di modifica. Queste informazioni permettono un'analisi più precisa di quando è stata eseguita una modifica o un inserimento in una determinata tabella.

Le tabelle di business sono 5. Si indicherà come PK il campo o il campi che formano la chiave primaria e FK eventuali chiavi esterne. Anche se all'atto pratico la chiave primaria e gli indici sono stati impostati sugli Id, in quanto per la logica della soft delete è possibile avere la presenza di più tuple uguali, PK indicherà la tupla usata per i controlli lato backend per i campi che forniscono univocità al record non cancellato.

La tabella di anagrafica clienti (Customers) prevede semplicemente due colonne VARCHAR, il codice (PK) e la descrizione in quanto sono le uniche informazioni necessarie per l'applicazione dato che il resto dei dati del cliente è mantenuto nel server interno di GP9Over.

La tabella di anagrafica versioni (Versions) prevede due colonne VARCHAR, nome di versione e descrizione, e tre colonne INT che indicano i numeri di versione, major, minor e build number; la tripla che formano questi numeri è la PK della tabella.

La tabella di anagrafica moduli (Modules) prevede una coppia di colonne testuali gemelle, il codice e la descrizione tecnica e il codice e la descrizione commerciale. Il codice tecnico (PK) corrisponde allo stesso codice che viene utilizzato su GPOne per la gestione dei permessi e delle abilitazioni mentre il codice commerciale serve solo ai commerciali qualora il nome di un modulo fosse troppo tecnico per essere intuitivo per l'utilizzo da cliente. Oltre a questi, sono presenti una colonna INT per fornire un raggruppamento sui moduli, utile nell'interfaccia grafica per avere dei pannelli espandibili con i moduli della stessa categoria, e una colonna booleana per indicare se il modulo è uno dei moduli che permettono un numero di abilitazioni. L'ultima colonna della tabella è una FK alla tabella delle versioni e serve per indicare qual'è la versione minima in cui viene gestito il modulo. Questo è utile nella creazione di una licenza per eventualmente disabilitare la scelta di alcuni moduli qualora la versione per cui si sta inserendo la licenza sia troppo vecchia.

La tabella delle licenze è chiaramente la più importante a livello di business. In questa tabella si iniziano a usare anche le entità di auditing. A differenza delle altre tabelle, in questo caso alle date di creazione, modifica e cancellazione sono anche associate delle FK alla tabella utenti (Users). Sono presenti due ulteriori colonne datetime per le date di approvazione licenza e di blocco licenza, con le relative FK agli utenti che hanno compiuto l'azione. Sono presenti le FK relative alla versione e al cliente; la coppia delle due FK fornisce anche la PK della tabella. Infine sono presenti una data di scadenza (che può essere NULL), una colonna testuale di chiave di attivazione, una di identificativo del server cliente e una colonna booleana che permette di evitare il controllo di versione su GPOne per sviluppatori e tester.

L'ultima tabella delle entità di business è una molti a molti che collega licenze e moduli. Questa tabella contiene le due colonne FK di licenza e modulo, che formano la PK, una colonna INT che indica il numero di abilitazioni (nullabile), se il modulo collegato ha il flag per questa casistica attivo, e la data di scadenza (nullabile).

3.1.2 Tabelle di audit

Le tabelle di audit sono strettamente correlate agli utenti e ai gruppi che vengono usati nel tenant di Azure di OSL.

La tabella utenti (Users) presenta come colonne il nome completo e la mail (PK) dell'utente, oltre a due colonne datetime per monitorarne il primo e ultimo accesso.

La tabella dei gruppi (Groups) in questo caso presenta solo le colonne del nome del gruppo e il GUID associato ad esso nel tenant di Azure.

Tra le due tabelle appena descritte c'è anche un collegamento con la tabella UsersGroups, che è una molti a molti standard.

Gli utenti vengono censiti nella tabella al primo accesso che fanno sull'applicazione, l'informazione viene presa dalla sessione del single sign on di Azure. I gruppi invece sono stati censiti a priori dato che censirli al momento dell'accesso forniva solo il GUID del gruppo, ma non il nome.

La tabella log di audit (AuditLogs) è probabilmente seconda sola alla tabella delle licenze per importanza. Le colonne di questa tabella permettono di risalire a quale entità è collegata un'azione, da chi è stata effettuata e quando. Sono dunque presenti una FK agli utenti, oltre a delle colonne stringa per il tipo di entità per cui viene inserito il log, il tipo di azione, i cambiamenti che sono stati

fatti (sotto forma di JSON), un identificativo della sessione e una colonna datetime per la data ora di quando è stato creato il log.

Questa tabella viene popolata con ogni azione che viene eseguita sulle licenze o sui moduli.

3.1.3 Utilizzo di Entity Framework per gestire il database

Si vedrà una parte del codice che permette di gestire il database e le sue entità. Si è già accennato nei capitoli precedenti alla tecnologia backend di EF Core. Nel progetto EF Core ha permesso di impostare tutte le tabelle, i vincoli, le chiavi e gli indici direttamente da codice. Sviluppare con il framework è abbastanza semplice.

L'oggetto che permette il tutto è il DbContext. È possibile estendere la classe di EF Core creando il proprio DatabaseContext come fatto nel listing 3.1; come esempio più semplice possibile è stato aggiunto un solo DbSet relativo all'entità Customer. Il DbSet rappresenta una tabella del database. Nella funzione OnModelCreating vengono poi richiamate tutte le azioni che vanno fatte per le tabelle del database, in questo caso si è sfruttato una funzione del della classe ModelBuilder che permette di andare a ritrovare tutte le classi che implementano una certa interfaccia e di richiamarne il metodo Configure. La parte di codice relativa al metodo OnConfiguring serve per aggiungere alcune funzionalità aggiuntive al DatabaseContext, in questo caso l'unica aggiunta è stata riguardo un interceptor per inserire un log al momento del salvataggio di un operazione a database.

Codice 3.1: Codice DatabaseContext

```
1  using Microsoft.EntityFrameworkCore;
2  using Microsoft.Extensions.DependencyInjection;
3  namespace Genba.CloudLicenses.Infrastructure.Sql;
4  public class DatabaseContext : DbContext
5  {
6      public DatabaseContext(DbContextOptions<DatabaseContext> options) : base(
           options)
7      {
8      }
9      public DbSet<Customer> Customers => Set<Customer>();
10     protected override void OnConfiguring(DbContextOptionsBuilder
           optionsBuilder)
```

```
11      {
12          if (_serviceProvider != null)
13          {
14              var currentUserService = _serviceProvider.GetService<
                    ICurrentUserService>();
15              if (currentUserService != null)
16              {
17                  optionsBuilder.AddInterceptors(new AuditSaveChangesInterceptor
                        (currentUserService));
18              }
19          }
20      }
21      protected override void OnModelCreating(ModelBuilder modelBuilder)
22      {
23          modelBuilder.ApplyConfigurationsFromAssembly(typeof(DatabaseContext).
                Assembly);
24      }
25  }
```

Nel listing 3.2 si può osservare una delle classi che viene usata per impostare la tabella a database. L'interfaccia che viene estesa e presa dal metodo usata in OnModelCreating è IEntityTypeConfiguration<T>, con T che indica il tipo della classe della tabella. L'esempio riportato della classe AuditLog (con meno metodi dell'implementazione effettiva) permette di osservare una serie di funzioni di EF.

- il metodo HasKey, per indicare la chiave primaria di un entità;
- il metodo Property, per eseguire operazioni su una colonna specifica, che poi può a sua volta richiamare altri metodi come IsRequired o HasMaxLength;
- il metodo HasIndex, per dichiarare degli indici sulla tabella;
- i metodi hasOne e hasMany, per dichiarare le chiavi esterne su un entità e il tipo di relazione tra due tabelle, oltre a poter impostare il comportamento in caso di cancellazione (in questo caso non va a cancellare a cascata i figli di un record dato che il comportamento è impostato su Restrict).

Codice 3.2: Codice Builder Extension AuditLog

```
using Genba.CloudLicenses.Domain.Model;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;
namespace Genba.CloudLicenses.Infrastructure.Sql.Entities.AuditLogs;
public class AuditLogModelBuilderExtension : IEntityTypeConfiguration<AuditLog
    >
{
    public void Configure(EntityTypeBuilder<AuditLog> builder)
    {
        builder.HasKey(x => x.Id);
        builder.Property(x => x.Id)
            .ValueGeneratedOnAdd();

        builder.Property(x => x.EntityType)
            .IsRequired()
            .HasMaxLength(100);

        builder.Property(x => x.Timestamp)
            .IsRequired()
            .HasDefaultValueSql("GETUTCDATE()");

        builder.HasIndex(x => new { x.EntityType, x.EntityId })
            .HasDatabaseName("IX_AuditLogs_EntityType_EntityId");

        builder.HasOne(x => x.User)
            .WithMany()
            .HasForeignKey(x => x.UserId)
            .OnDelete(DeleteBehavior.Restrict);
    }
}
```

Nel listing 3.3 è presente infine il pezzo di codice necessario per iniettare il database context nella dependency injection dell'applicazione .NET nel file "Program.cs". Il metodo GetConnectionString va a leggere il file di configurazione (solitamente per le applicazione .NET è "appsettings.json") in una sezione specifica "ConnectionStrings", in cui è presente la stringa di connessione al database.

Codice 3.3: Codice Dependency Injection DatabaseContext

```
builder.Services.AddDbContext<DatabaseContext>((serviceProvider,
    optionsBuilder) =>
{
    var connectionString = builder.Configuration.GetConnectionString("
        DatabaseContext");
    optionsBuilder
        .UseSqlServer(connectionString);
});
```

Le modifiche al database vengono fatte con quelle che vengono chiamate migrations; nel codice vengono generati dei file che contengono tutte le modifiche da fare al database, insieme a un file di snapshot della struttura del database che viene aggiornato dopo ogni migration. Questi file vengono generati con il primo comando dello script nel listing 3.4, e vengono applicati a database con il secondo comando.

Codice 3.4: Script di generazione migrations e aggiornamento database

```
dotnet ef migrations add 1.0 --project .\Genba.CloudLicenses.Infrastructure

dotnet ef database update --project .\Genba.CloudLicenses.Infrastructure --
```

3.2 Struttura API BE

La struttura della soluzione backend, come si è già detto, segue la clean architecture. Nella sezione precedente si è già accennato alla parte di dominio e di infrastruttura. Rimangono dunque da esplorare il livello dell'API vera e propria ed il livello application. Si userà come esempio la classe Customer; i pezzi di codice riportati non conterranno tutti i metodi di crud o le query implementate ma solo quelli necessari per la spiegazione.

Il punto d'ingresso è dato dalla classe controller dell'entità che si occupa di esporre le chiamate REST. Le librerie messe a disposizione da Microsoft ASP.NET permettono di gestire in modo semplice i metodi REST standard e i codici di ritorno. Nel listing 3.5 è possibile osservare alcuni elementi d'interesse:

- l'attributo ApiController, assegnato alla classe CustomerController, viene sfruttato in sede di dependency injection, attraverso una funzione ("builder.Services.AddControllers()") che dinamicamente registra tutte le classi con quell'attributo senza dichiararle esplicitamente in dependency injection;
- le due chiamate GET e POST, che richiamano metodi di ASP.NET per gestire i codici di ritorno (400 per BadRequest, 200 per la risposta della query e 201 per l'inserimento della risorsa, 404 per il NotFound);
- le classi specifiche per Request e Response, i DTO (Data Type Object) di interfacciamento con il frontend, che vengono poi mappate sulla classe di dominio.

Codice 3.5: Classe controller

```
using Genba.CloudLicenses.Application;
using Genba.CloudLicenses.Application.Customers;
using Microsoft.AspNetCore.Mvc;
namespace Genba.CloudLicenses.Api.Customers;
[ApiController]
public class CustomersController(
    ILogger<CustomersController> logger,
    ICustomerService customerService) : ControllerBase
{
    private readonly ILogger<CustomersController> _logger = logger;
    private readonly ICustomerService _customerService = customerService;
    private const string ApiBase = "api/customers";
    [HttpGet($"{ApiBase}/{{id}}")]
    [ProducesResponseType(typeof(CustomerResponse), StatusCodes.Status200OK)]
    public IActionResult GetById([FromRoute] int id)
    {
        try
        {
            var customer = _customerService.GetById(id);
            if (customer is null)
            {
                return NotFound();
```

```
23              }
24
25              var customerResponse = customer.MapToResponse();
26              return Ok(customerResponse);
27          }
28          catch (Exception ex)
29          {
30              _logger.LogError(ex, ex.Message);
31              return BadRequest(ex.Message);
32          }
33      }
34      [HttpPost(ApiBase)]
35      [ProducesResponseType(typeof(CustomerResponse), StatusCodes.
            Status201Created)]
36      public IActionResult Insert([FromBody] CustomerRequest customerRequest)
37      {
38          try
39          {
40              var customer = customerRequest.MapToDomain();
41              _customerService.Insert(customer);
42              var customerResponse = customer.MapToResponse();
43
44              return CreatedAtAction(nameof(GetById), new { id = customer.Id },
                    customerResponse);
45          }
46          catch (Exception ex)
47          {
48              _logger.LogError(ex, ex.Message);
49              return BadRequest(ex.Message);
50          }
51      }
52  }
```

Tutte le funzioni dei vari controller si occupano solo del mapping dei DTO nelle entità di dominio e viceversa; l'effettiva logica di business è implementata nelle classi Service delle varie entità. Nell'applicazione non sono presenti particolari logiche di business se non nel servizio

relativo alle licenze; i servizi tendono a fare da passaggio intermedio tra i controllers e i repositories (listing 3.6.

Codice 3.6: Classe service

```
1  using Genba.CloudLicenses.Application.Interfaces;
2  using Genba.CloudLicenses.Application.Validators;
3  using Genba.CloudLicenses.Domain.Model;
4  namespace Genba.CloudLicenses.Application.Customers;
5  public interface ICustomerService
6  {
7      Customer? GetById(int id);
8      Customer Insert(Customer customer);
9  }
10 public class CustomerService(
11     ICustomerRepository customerRepository,
12     INewEntityValidator<Customer> newValidator) : ICustomerService
13 {
14     private readonly ICustomerRepository _customerRepository =
           customerRepository;
15     private readonly INewEntityValidator<Customer> _newValidator =
           newValidator;
16
17     public Customer GetById(int id)
18     {
19         return _customerRepository.GetById(id) ?? Customer.Create(0, "EMPTY",
               "EMPTY");
20     }
21
22     public Customer Insert(Customer customer)
23     {
24         var validationResult = _newValidator.Validate(customer);
25         if (!validationResult.IsValid)
26         {
27             throw new Exception(validationResult.Errors[0].ErrorMessage);
28         }
29
```

```
30          return _customerRepository.Insert(customer);
31      }
32  }
```

Nei servizi vengono comunque sfruttati una serie di validatori prima di chiamare le funzioni dei repositories per tutte le chiamate di inserimento, modifica e cancellazione. Queste classi estendono un'interfaccia "IValidator" messa a disposizione dalla libreria FluentValidation e ritornano un oggetto ValidationResult; la libreria permette con una serie di metodi di poter impostare dei controlli in modo intuitivo sui vari attributi dell'entità (listing 3.7.

Codice 3.7: Validatore entità Customer

```
1  using FluentValidation;
2  using Genba.CloudLicenses.Application.Interfaces;
3  using Genba.CloudLicenses.Application.Validators;
4  using Genba.CloudLicenses.Domain.Model;
5  namespace Genba.CloudLicenses.Application.Customers.Validators;
6  public interface INewEntityValidator<T> : IValidator
7  {
8      ValidationResult Validate(T instance);
9  }
10 public class NewCustomerValidator : AbstractValidator<Customer>,
       INewEntityValidator<Customer>
11 {
12     public NewCustomerValidator(ICustomerRepository customerRepository)
13     {
14         RuleFor(customer => customer.Code)
15             .Must((customer, code) => customerRepository.GetByCode(code) ==
                   null)
16             .WithMessage(customer => "TRANSLATIONS.ERRORS.DUPLICATE_CODE");
17     }
18 }
```

Nella discesa dei vari strati del backend, l'ultimo è dato dalle classi dei repositories delle entità. Queste classi si occupano dell'interfacciamento vero e proprio con il database, infatti sono le uniche che chiamano direttamente il DatabaseContext per effettuare queries e operazioni sui records. Nel listing 3.8 si può osservare come sia il metodo di query sia quello di inserimento facciano riferimento

al DbSet dei Customers registrato come attributo all'interno del DatabaseContext; l'oggetto al programmatore appare come un oggetto Iterable e tramite Linq è possibile eseguire tutte le operazioni di filtro e ordinamento standard. I record del DbSet vengono istanziati solo quando viene usato un metodo per trasformare il risultato in un oggetto concreto, come il metodo FirstOrDefault usato nel metodo GetById, oppure con altri metodi come ToList o ToArray. L'inserimento o la cancellazione vengono fatti con metodi simili ai metodi di una lista, come si può vedere nel metodo Insert a cui viene aggiunto un oggetto Customer con il metodo Add. L'effettiva scrittura delle modifiche a database viene eseguita dal metodo SaveChanges.

Codice 3.8: Classe repository

```
using Genba.CloudLicenses.Application.Interfaces;
using Genba.CloudLicenses.Domain.Model;
using Microsoft.EntityFrameworkCore;
namespace Genba.CloudLicenses.Infrastructure.Sql.Repositories;
public class CustomerRepository(
    DatabaseContext databaseContext) : ICustomerRepository
{
    private readonly DatabaseContext _databaseContext = databaseContext;

    public Customer? GetById(int id)
    {
        return _databaseContext
            .Customers
            .Where(x =>
                x.Id == id &&
                !x.DeletedDateTime.HasValue)
            .FirstOrDefault();
    }
    public Customer Insert(Customer customer)
    {
        customer.CreationDateTime = DateTime.UtcNow;
        _databaseContext.Customers.Add(customer);
        _databaseContext.SaveChanges();

        return customer;
```

```
26      }
27  }
```

3.2.1 Generazione degli audit logs

Normalmente la chiamata della SaveChanges del DatabaseContext va direttamente a scrivere le modifiche sul database. Tuttavia è possibile mettersi nel mezzo di questa operazione per effettuare eventuali operazioni aggiuntive. Si è visto nella descrizione del DatabaseContext si è aggiunto nella configurazione un interceptor; in questo oggetto ha il compito di intercettare la chiamata SaveChanges e in questo caso di andare a prendere i record inseriti o modificati nel DatabaseContext e decidere se inserirne un log. Questa operazione, in base al requisito fuzionale RF06, almeno per questa prima iterazione non dev'essere eseguita per ogni tipo di tabella, ma solo per quelle relative alle licenze e ai moduli.

Codice 3.9: Classe interceptor

```
1  using Genba.CloudLicenses.Application;
2  using Genba.CloudLicenses.Domain.Model;
3  using Genba.CloudLicenses.Infrastructure.Sql.Audit;
4  using Microsoft.EntityFrameworkCore;
5  using Microsoft.EntityFrameworkCore.ChangeTracking;
6  using Microsoft.EntityFrameworkCore.Diagnostics;
7  using System.Text.Json;
8  namespace Genba.CloudLicenses.Infrastructure.Sql;
9  public class AuditSaveChangesInterceptor(ICurrentUserService
       currentUserService) : ISaveChangesInterceptor
10 {
11     private readonly ICurrentUserService _currentUserService =
           currentUserService;
12     public InterceptionResult<int> SavingChanges(
13         DbContextEventData eventData,
14         InterceptionResult<int> result)
15     {
16         if (eventData.Context is not DatabaseContext context) return result;
17
```

```
18          UpdateAuditableEntities(context);
19          return result;
20      }
21      private void UpdateAuditableEntities(DatabaseContext context)
22      {
23          var auditEntries = GetAuditEntries(context);
24
25          foreach (var auditEntry in auditEntries)
26          {
27              context.Set<AuditLog>().Add(auditEntry);
28          }
29      }
30      private List<AuditLog> GetAuditEntries(DatabaseContext context)
31      {
32          var auditEntries = new List<AuditLog>();
33          var userId = _currentUserService.GetUserId();
34          var userEmail = _currentUserService.GetUserEmail();
35          var sessionId = _currentUserService.GetSessionId();
36          if (!userId.HasValue || string.IsNullOrEmpty(userEmail))
37              return auditEntries;
38          var tracker = context.ChangeTracker;
39          var changedEntities = tracker.Entries()
40              .Where(e => e.State == EntityState.Added ||
41                          e.State == EntityState.Modified ||
42                          e.State == EntityState.Deleted)
43              .ToList();
44          foreach (var entry in changedEntities)
45          {
46              var strategy = GetStrategyForType(entry.Entity.GetType());
47              if (strategy == null || strategy.ShouldSkip(entry))
48                  continue;
49              var auditLog = strategy.CreateAuditLog(entry, userId.Value,
                    userEmail, sessionId);
50              if (auditLog != null)
51              {
52                  auditEntries.Add(auditLog);
```

```
53              }
54          }
55          return auditEntries;
56      }
57      private static IAuditStrategy? GetStrategyForType(Type entityType) =>
            entityType switch
58      {
59          Type t when t == typeof(Domain.Model.License) => new
                LicenseAuditStrategy(),
60          Type t when t == typeof(LicenseModule) => new
                LicenseModuleAuditStrategy(),
61          Type t when t == typeof(Module) => new ModuleAuditStrategy(),
62          _ => null
63      };
64  }
```

Nel listing 3.9 si possono osservare tre punti di interesse:

- l'utilizzo di ICurrentUserContext, un'interfaccia che fornisce un wrapper per l'interfaccia IHttpContextAccessor, che per best practice non si passa direttamente al costruttore, che permette di accedere alle informazioni dell'utente loggato;
- l'utilizzo di strategie diverse legate al tipo di entità che si sta modificando, in questo modo è possibile personalizzare la generazione dei log in base all'entità;
- l'utilizzo del ChangeTracker per poter riconoscere i record Tracked che nel DatabaseContext indica quelli che hanno subito delle modifiche.

L'approccio seguito per questa funzionalità è stato di mettere il tutto all'interno del codice dell'applicazione backend. Tuttavia era possibile anche esternare l'interceptor in stile serverless. Essendo nell'ecosistema Azure, si potevano anche utilizzare le Azure Functions, che si collegavano come un trigger a ogni operazione effettuata al database di Azure SQL Server. Si è scelto di non utilizzare quest'approccio non per una questione di costi, dato che Azure fornisce una soglia di attivazioni mensile molto alta che difficilmente si supera se non si ha un'applicazione ad alto traffico, per non utilizzare subito troppi servizi di Azure, dato che la maggior parte del team di sviluppo non

ha ancora le competenze ne per sfruttare al meglio le risorse di Azure ne per la programmazione serverless.

3.2.2 Chiamate per GP9Over

Nel requisito funzionale RF04 è stato richiesto che l'integrazione con GP9Over fosse bidirezionale. Per quanto riguarda la comunicazione da GP9Over, le chiamate sfruttate sono le stesse utilizzate dal client dell'applicazione licenze; dunque vengono utilizzate le chiamate API standard esposte per i clienti e per le licenze. Chiaramente viene fatto prima una chiamata di controllo, che in questo caso sono delle semplici GET sia sui clienti che sulle licenze, per verificare che le informazioni non siano state già inserite o che necessitino di aggiornamento.

La comunicazione opposta dall'applicazione licenze a GP9Over viene comunque eseguita da GP9Over. Si è scelto di demandare al gestionale interno il compito di eseguire una chiamata periodica all'applicazione per ottenere tutte le modifiche che sono state effettuate dall'ultima chiamata. L'implementazione della chiamata sul gestionale interno è stata demandata al reparto programmazione GP9Over e dunque non verrà approfondita.

Codice 3.10: Chiamata per GP9Over e oggetto di ritorno

```
1  [HttpGet($"{ApiBase}/getforgp9over")]
2  [ProducesResponseType(typeof(Gp9OverCheckResponse), StatusCodes.Status200OK)]
3  public IActionResult GetModifiedRecordsForGp9Over([FromQuery] DateTime
       lastReadDateTime)
4  {
5      try
6      {
7          var response = _licenseService.GetModifiedEntitiesForGp9Over(
               lastReadDateTime);
8          var mappedResponse = response.MapToResponse();
9          return Ok(mappedResponse);
10     }
11     catch (Exception ex)
12     {
13         _logger.LogError(ex, ex.Message);
14         return BadRequest(ex.Message);
```

```
15      }
16  }
17  public class Gp9OverCheckResponse
18  {
19      public List<CustomerResponse> ModifiedCustomers { get; set; }
20      public List<LicenseResponse> ModifiedLicenses { get; set; }
21  }
```

Sull'applicazione licenze è stata esposta una chiamata nel controller delle licenze che prende in input l'ultima data ora in cui è stata effettuata la chiamata dal gestionale. Nel listing 3.10 si può vedere che viene richiamata una funzione nel LicenseService, in cui fa due query in sequenza, una relativa ai clienti e una relativa alle licenze, con gli eventuali moduli. Entrambe queste query fanno un filtro di maggiore uguale su tutte le date delle entità, quindi data di creazione, modifica e cancellazione, e per le licenze e la molti a molti LicenseModule anche le date di attivazione e di blocco. L'oggetto di risposta alla chiamata di GP9Over è semplicemente un insieme di liste relative ai clienti e alle licenze. La chiamata viene eseguita periodicamente almeno una volta al giorno, ma il tempo di attesa tra una chiamata e l'altra è impostabile a piacere su GP9Over.

3.3 Frontend

La struttura dell'interfaccia frontend non è particolarmente complessa. Questo è dovuto al fatto che il numero di interfacce è molto limitato, e ad eccezione delle schermate relative alle licenze, la maggior parte sono composte da griglie o da forms.

Una necessità che si è presentata in fase di impostazione della soluzione e del codice frontend è stata quella relativa a quali componenti utilizzare. La scelta poteva essere una tra l'implementare dei componenti ad hoc per l'applicazione licenze oppure usufruire di pacchetti di componenti già realizzati. Si è scelto di perseguire la seconda strada. Purtroppo non è stato possibile riutilizzare i componenti di UI di GPOne, poichè prevedono alcune strutture e servizi che sono presenti nell'infrastruttura client del gestionale che avrebbero introdotto solo ulteriore complessità in un'applicazione che per il momento doveva essere il più semplice possibile. In un primo momento si è pensato di utilizzare il kit di componenti di Angular Material, già usato in passato anche per alcuni sviluppi sulla suite di GPOne; tuttavia anche con alcuni stakeholders si è deciso di valutare un altro

pacchetto graficamente diverso. La scelta finale è ricaduta sull'utilizzo dei componenti di PrimeNG. PrimeNG è un pacchetto di componenti in parte open-source realizzato da PrimeTek. Il pacchetto offre una serie di templates preimpostati da cui poter partire con lo sviluppo. Ad eccezione di un template, tutti sono a pagamento. Il template open-source Sakai è stato quello utilizzato per sviluppare l'interfaccia dell'applicazione licenze (figura 3.2). Tutti i componenti utilizzati sono basati su Sakai, con l'unica modifica legata agli stili CSS utilizzati.

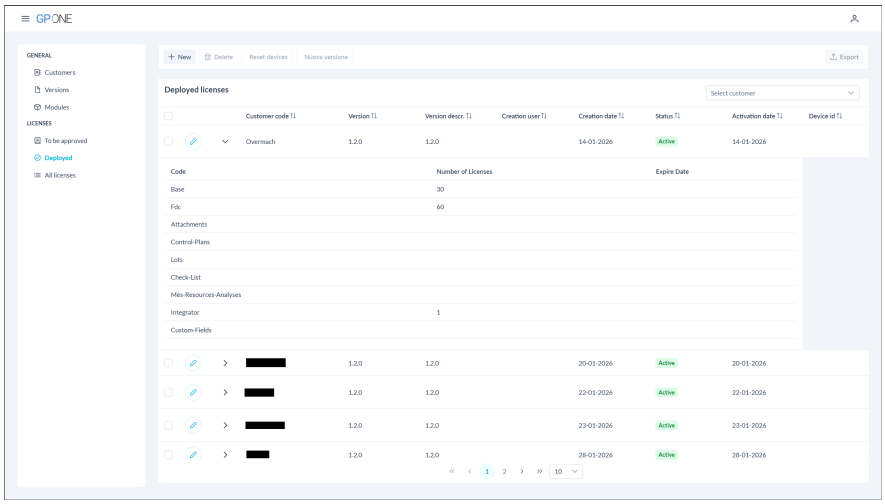

Figura 3.2: Interfaccia griglia licenze dell'applicazione in cloud

3.3.1 Utilizzo di signals e observables

Una differenza importante tra lo sviluppo di questo client e quello realizzato su GPOne è l'utilizzo dei signals.

Si è accennato nei capitoli precedenti come su GPOne siano state usate le promises. Le promises sono degli oggetti con cui vengono gestite le chiamate asincrone su Angular; queste sono state usate principalmente per gestire la comunicazione client-server. Il grosso problema di questi oggetti è che, una volta eseguita una chiamata con una promise, non sono più cancellabili e inoltre possono ritornare un solo valore dalla chiamata. Questo causava spesso il blocco dell'unico thread dedicato al client, poichè costretto a rimanere in attesa se usato con un operatore await.

L'evoluzione di questo approccio è stato il paradigma di reactive programming; con l'obiettivo di non bloccare il thread, l'asincronia viene gestita tramite degli oggetti chiamati observables. Gli observables vengono gestiti come un flusso (stream) di eventi o notifiche, simili a un iterable, ma

con la differenza che questi vengono popolati in modo asincrono nel tempo. Questo risolve entrambi i problemi causati dalle promises, poichè non si rimane in attesa di una risposta dalla chiamata ma semplicemente ci si mette in ascolto di eventi come nel paradigma event-driven; inoltre per il modo in cui sono fatti gli observables non solo è possibile ricevere più valori di ritorno, ma interrompere in qualsiasi momento il flusso di dati dell'observable. In Angular la libreria che implementa gli observables è RxJS.

Le promises inoltre si basano sulla "Zone.js", una libreria di Angular che gestisce tutti gli aggiornamenti della UI in automatico, togliendo allo sviluppatore la necessità di dover richiamare manualmente un aggiornamento dei componenti grafici. L'utilizzo della zona introduce però un overhead più che significativo nelle applicazioni particolarmente estese con molti componenti, a causa di tutto l'aggiornamento dell'albero dei componenti grafici. GPOne soffre di questo in quanto è un'applicazione con un numero molto alto di componenti in un singolo client. La direzione che Angular ha preso negli ultimi anni è la "zoneless", ovvero il non usare più la libreria in quanto per le applicazioni moderne introduceva colli di bottiglia nelle prestazioni. È grazie a questo cambio di paradigma che sono stati sviluppati i signals.

I signals sono stati introdotti in Angular dalla versione 17, e sono dei contenitori di valori che notificano qualsiasi cambiamento del proprio stato interno a chiunque stia utilizzando quel componente. Questo evita l'aggiornamento globale dell'albero dei componenti e va a notificare ai soli componenti del DOM che utilizzano quel componente l'avvenuto cambiamento, quindi non verrà aggiornata l'intera applicazione ma solo una piccola parte. Ne consegue una pesante riduzione del collo di bottiglia dovuto all'aggiornamento dell'albero.

```
1  export class ModulesListComponent {
2    moduleService = inject(ModulesService);
3
4    selectedModules: Module[] | undefined;
5    moduleDialog = signal(false);
6    modules = this.moduleService.availableModules;
7    loadingState = this.moduleService.loadingState;
8
9    singleSelectedModule = signal<Module>({} as Module);
10
11   selectionChanged(modules: Module[]) {
```

```
    this.moduleService.selectedModules.set(modules);
  }

  closeDialog() {
    this.moduleDialog.set(false);
  }

  editModule(module: Module) {
    this.singleSelectedModule.set(module);
    this.moduleDialog.set(true);
  }
}
```

Codice 3.11: Esempio di codice dei signals nel frontend

La parte relativa al servizio delle chiamate all'API backend è stata realizzata utilizzando gli observables. Si è però deciso di sfruttare un software esterno open-source per la generazione automatica del codice di questo servizio. Nswag è un tool di terze parti che permette di generare il codice client o server a partire da un file JSON di specifica Swagger/OpenAPI. Questo evita tutta la complessità di implementare le chiamate esposte da un endpoint REST e tutti gli errori causati da eventuali errori umani nel mappare le chiamate.

```
export const API_BASE_URL = new InjectionToken<string>('API_BASE_URL');

@Injectable({
  providedIn: 'root',
})
export class ApiBase {
  private http: HttpClient;
  private baseUrl: string;
  protected jsonParseReviver: ((key: string, value: any) => any) | undefined =
    undefined;

  constructor(
    @Inject(HttpClient) http: HttpClient,
    @Optional() @Inject(API_BASE_URL) baseUrl?: string
  ) {
```

```
    this.http = http;
    this.baseUrl = baseUrl ?? '';
  }

  customers_GetAll(): Observable<CustomerResponse[]> {
    let url_ = this.baseUrl + '/api/customers';
    url_ = url_.replace(/[?&]$/, '');

    let options_: any = {
      observe: 'response',
      responseType: 'blob',
      headers: new HttpHeaders({
        Accept: 'text/plain',
      }),
    };

    return this.http
      .request('get', url_, options_)
      .pipe(
        _observableMergeMap((response_: any) => {
          return this.processCustomers_GetAll(response_);
        })
      )
      .pipe(
        _observableCatch((response_: any) => {
          if (response_ instanceof HttpResponseBase) {
            try {
              return this.processCustomers_GetAll(response_ as any);
            } catch (e) {
              return _observableThrow(e) as any as Observable<
                CustomerResponse[]
              >;
            }
          } else
            return _observableThrow(response_) as any as Observable<
              CustomerResponse[]
```

```
52              >;
53          })
54      );
55  }
```

Codice 3.12: Esempio servizio API autogenerato da NSwag

L'utilizzo di observables e signals nella soluzione dell'applicazione delle licenze ha permesso di esplorare questi concetti per la prima volta nella ricerca e sviluppo. Questo permetterà nel (breve) futuro di poter iniziare a convertire tutte le chiamate al server fatte tramite le promises e l'abbandono dell'utilizzo della zona nei client di GPOne.

3.4 Integrazione con Azure SSO

L'utilizzo che è stato fatto di Azure non si è limitato allo sfruttamento delle infrastrutture in cloud, ma è parte integrante del codice dell'applicazione. Sfruttare il SSO di Azure ha permesso di raggiungere in un colpo solo due requisiti, il requisito RF02 legato all'autenticazione con mail aziendale e il requisito RF03 legato alla gestione dei permessi.

Avendo già utilizzato per la gestione del progetto di GPOne Azure Devops e utilizzando la suite di Office365 e Sharepoint, la maggior parte degli utenti dell'azienda era già stata inserita in un gruppo di utenti allineato con il proprio reparto di appartenenza. Questo ha permesso di non dover gestire il censimento degli utenti o l'assegnazione nei vari gruppi. A livello di codice si sono dovuti integrare l'interceptor dell'autenticazione Microsoft, i dati del tenant dell'azienda di Entra ID.

3.4.1 Integrazione backend

La maggior parte delle modifiche sono state fatte sul progetto dell'API. I gruppi degli utenti sono stati censiti manualmente sul database. Nel listing 3.13 si possono osservare i passaggi per impostare sia la parte di autenticazione che di autorizzazione con Azure. Per la parte di autenticazione, utilizzando la libreria "Microsoft.Identity.Web" e l'autenticazione JWT, basta richiamare il metodo "AddMicrosoftIdentityWebApi"; il metodo prende in input un dizionario chiave-valore, in questo caso viene letto dal file "appsettings.json". Il listing 3.14 mostra la sezione dell'appsettings che contiene le impostazioni per l'autenticazione Azure: l'url dell'istanza, solitamente identica per tutte

le autenticazioni Azure, e i due id di tenant, l'id del tenant all'interno di Entra ID, e il client, in questo caso l'id della static web app all'interno di Entra ID, e non l'id della risorsa in se nell'infrastruttura Azure..

Codice 3.13: Parte di codice di startup per impostare Azure SSO

```
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme).
    AddMicrosoftIdentityWebApi(builder.Configuration.GetSection("AzureAd"));

builder.Services.AddAuthorizationBuilder().AddPolicy(PolicyGroups.LicenseAdmin
    , policy=> policy.RequireClaim("groups", PolicyGroups.AdminGroupId))
.AddPolicy(PolicyGroups.LicenseEditor, policy =>policy.RequireAssertion(
    context =>context.User.HasClaim("groups", PolicyGroups.AdminGroupId) ||
    context.User.HasClaim("groups", PolicyGroups.UserGroupId))).AddPolicy(
    PolicyGroups.LicenseReader, policy =>policy.RequireAssertion(context =>
    context.User.HasClaim("groups", PolicyGroups.AdminGroupId) ||context.User.
    HasClaim("groups", PolicyGroups.UserGroupId) ||context.User.HasClaim("
    groups", PolicyGroups.ReaderGroupId)));
```

Codice 3.14: Impostazioni Azure SSO in appsettings

```
"AzureAd": {
    "Instance": "https://login.microsoftonline.com/",
    "TenantId": "GUID tenant",
    "ClientId": "GUID client"
  }
```

Per quanto riguarda la parte di autorizzazione, sono state introdotte per il momento tre policies diverse per la gestione dei permessi:

- la policy per gli admin, che possono effettuare ogni azione sull'applicazione;
- la policy degli editor, per quegli utenti che possono inserire ed approvare licenze, quindi in riferimento a quanto detto nelle sezioni precedenti sarebbe il gruppo di commerciali e analisti;
- la policy dei lettori, per gli utenti che possono solo leggere o scaricare una licenza, quindi i consulenti.

L'inserimento di una policy è anch'essa semplice. Il controllo viene fatta sui claims contenuti nell'header della chiamata che viene fatta. Per il caso degli admin, la policy richiede esplicitamente il claim specifico che l'utente faccia parte del gruppo admin. Per le altre due, la condizione della policy è messa in OR logico, quindi basta che l'utente faccia parte di uno dei gruppi affinchè gli venga permesso l'utilizzo delle chiamate che richiedono quel grado di autorizzazione. Una volta registrati autenticazione e autorizzazione nello startup dell'applicazione, integrare i livelli di autorizzazione nelle chiamate API viene fatto utilizzando l'attributo "Authorize". Questo viene aggiunto tra gli attributi della chiamata HTTP e della risposta (listing 3.15). Se un utente facesse una chiamata per un endpoint per cui non ha i permessi, il framework gestisce autonomamente l'invio del codice di errore 401 Unauthorized.

Codice 3.15: Esempio di chiamata controller con autorizzazione

```
[HttpGet($"{ApiBase}/{{id}}")]
[Authorize(Policy = PolicyGroups.LicenseReader)]
[ProducesResponseType(typeof(LicenseResponse), StatusCodes.Status200OK)]
public IActionResult GetById([FromRoute] int id)
{
    try
    {
        var license = _licenseService.GetById(id);
        if (license is null)
        {
            return NotFound();
        }
        var customerResponse = license.MapToResponse();
        return Ok(customerResponse);
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, ex.Message);
        return BadRequest(ex.Message);
    }
}
```

3.4.2 Integrazione frontend

Anche per la parte su Angular è necessario sfruttare alcune librerie Microsoft. Le operazioni da fare sono in parte simili al backend. È necessario mappare le informazioni relative al tenant e al client di Entra Id, ed è necessario impostare la comunicazione con Azure al fine di ottenerne il token di autenticazione da aggiungere all'header delle chiamate che vengono fatte all'API.

Come operazioni preliminari sono state definite una serie di costanti (listing **??** :

- l'interceptor AuthInterceptor, una funzione che funge da interceptor http e viene iniettato in dependency injection nel file "app.config.ts", e si occupa di intercettare le chiamate all'API e di aggiungerne nell'header la chiave di autorizzazione con il bearer token ottenuto da Azure;
- la costante authConfig, che contiene le informazioni da usare di tenant e client azure;
- l'interfaccia AuthConfig, che conterrà il riferimento all'istanza del client Microsoft da usare per l'autenticazione e, una volta valorizzati, i riferimenti al token e ai dati dell'account.

```
1  import { HttpInterceptorFn } from '@angular/common/http';
2  import { inject } from '@angular/core';
3  import { AuthService } from '../pages/auth-pages/auth.service';
4  import { environment } from '../../environments/environment';
5  import { AccountInfo, PublicClientApplication } from '@azure/msal-browser';
6  export const authInterceptor: HttpInterceptorFn = (req, next) => {
7    const authService = inject(AuthService);
8    const token = authService.userData().token;
9    let headers: Record<string, string> = {};
10   if (token) {
11     headers['Authorization'] = "Bearer ${token}";
12   }
13   const cloned = req.clone({ setHeaders: headers });
14   return next(cloned);
15 };
16 export const authConfig = {
17   auth: {
18     clientId: environment.clientId,
19     authority: "https://login.microsoftonline.com/${environment.tenantId}",
```

```
20      redirectUri: environment.redirectUri,
21    },
22  };
23  const data: AuthConfig = {
24    account: null as AccountInfo | null,
25    msalInstance: new PublicClientApplication(authConfig),
26    token: '',
27  };
28  export function useAuth(): AuthConfig {
29    return data;
30  }
31  export interface AuthConfig {
32    account: AccountInfo | null;
33    msalInstance: PublicClientApplication;
34    token: string;
35  }
```

Codice 3.16: Costanti interceptor e configurazioni frontend

L'oggetto "environment" su Angular ha lo stesso scopo di "appsettings.json" sul backend, quindi contiene tutte i valori delle impostazioni da utilizzare nell'applicazione frontend. Non si vedrà nel dettaglio la classe di AuthService, in quanto è un semplice contenitore per le informazioni dell'utente.

Questi step preliminari hanno permesso poi di impostare la vera e propria pagina di login (listing 3.17). Tuttavia non è stato implementato nessun template per la pagina in quanto il componente reindirizza alla pagina di login di Microsoft. Nel componente di login viene gestito tutto attraverso l'istanza del client di autenticazione Microsoft. Se è il primo login della sessione, la chiamata non ritornerà i dati di alcun account e dunque indirizzerà alla login Microsoft. Una volta loggati, vengono popolate nell'AuthService tutte le informazioni dell'account della sessione. Le informazioni vengono poi passate al backend con una chiamata che andrà a censire l'utente qualora fosse il primo accesso in assoluto all'applicazione, altrimenti verrà semplicemente aggiornata la sua data di ultimo accesso. Alla fine del processo di login, nel momento in cui il signal collegato al token nei dati utente salvati nell'AuthService viene popolato, l'applicazione reindirizza alla pagina radice dell'applicazione vera e propria.

```
import { Component, effect, inject } from '@angular/core';
import { Router } from '@angular/router';
import { AuthenticationResult } from '@azure/msal-browser';
import { ButtonModule } from 'primeng/button';
import { AuthService } from '../auth.service';
@Component({
  selector: 'app-login-page',
  imports: [ButtonModule],
  template: "",
  styleUrl: './login-page.scss',
})
export class LoginPage {
  private readonly authService = inject(AuthService);
  private readonly route = inject(Router);
  constructor() {
    effect(() => {
      if (this.authService.userData().token) {
        this.route.navigate(['/']);
      }
    });
  }
  async ngOnInit() {
    await this.authService.authConfig.msalInstance.initialize();
    await this.authService.authConfig.msalInstance
      .handleRedirectPromise()
      .then(async (authResult: AuthenticationResult | null) => {
        const accounts =
          this.authService.authConfig.msalInstance.getAllAccounts();
        if (accounts.length > 0) {
          this.authService.authConfig.account = accounts[0];
          const response: AuthenticationResult =
            await this.authService.authConfig.msalInstance.acquireTokenSilent
              ({
              account: this.authService.authConfig.account,
              scopes: [
                'api://GUID API/Versions.Read',
```

```
36              ],
37            });
38          this.authService.authConfig.token = this.getToken(response);
39          this.authService.setGroups(this.getGroups(response));
40          this.authService.setOperatorId(this.getOperatorId(response));
41          this.authService.setName(this.getName(response));
42          this.authService.setToken(this.getToken(response));
43          this.authService.setEmail(this.getEmail(response))
44          this.authService.sendUserData();
45        } else {
46          await this.authService.authConfig.msalInstance.loginRedirect();
47        }
48      });
49  }
50 }
```

Codice 3.17: Parte della classe del componente login

3.5 Codifica e decodifica file di licenza

L'implementazione della codifica e della decodifica della licenza va a toccare sia la parte dell'applicazione licenze che la parte dell'applicazione di GPOne.

Siccome il codice è in parte condiviso, lo sviluppo è stato messo in un progetto comune chiamato "Genba.License"; questo progetto è stato reso un pacchetto NuGet e inserito nel feed degli Azure Artifacts dell'azienda. In questo modo è possibile referenziare lo stesso codice da entrambi i repositories delle applicazioni. Il pacchetto viene ricompilato e rilasciato ogni volta che viene staccata una nuova versione di GPOne oppure ogni volta che viene aggiunto un nuovo modulo nella suite.

Il pacchetto contiene poche ma importanti classi condivise:

- le classi di modello per serializzare e deserializzare la licenza e i moduli;
- una classe statica contenente le funzioni per codificare/decodificare la licenza e per ottenere l'id del dispositivo d'installazione;

- una classe con le funzioni di lettura licenza, usata in questo caso solo da GPOne, ma che sfrutta le funzioni di decodifica nel pacchetto.

La codifica avviene in tre step (listing 3.18). La licenza viene serializzata in un array di bytes; questo è necessario in quanto le funzioni che andranno poi a eseguire la firma e la codifica lavorano su array di bytes. Dalla licenza serializzata e dalla chiave privata viene poi calcolata la firma. La chiave privata come già detto in precedenza, viene letta al momento del download di una licenza dall'Azure Key Vault. Per calcolare la firma si è usato il metodo dell'algoritmo di crittografia asimmetrica RSA; per l'algoritmo hash e lo schema di padding si è scelto di andare su due standard, lo SHA256 per il primo e il Pkcs1 per il secondo. Infine, ottenuta la firma digitale, la licenza e la firma vengono codificate in un'unica stringa in formato Base64. Questa stringa e la chiave pubblica vengono poi inserite in un oggetto, LicenseCertificated, che sarà poi il contenuto del file che verrà scaricato.

Codice 3.18: Codice per la codifica e la firma della licenza

```
1  public static LicenseCertificated GenerateAndSignLicense(LicenseContract
       license, string privateKeyPem, string publicKeyPem)
2  {
3      var licenseBytes = SerializeLicense(license);
4      var signature = SignLicenseData(licenseBytes, privateKeyPem);
5      var encodedLicense = EncodeLicense(signature, licenseBytes);
6      LicenseCertificated licenseCertificated = new()
7      {
8          LicenseEncoded = encodedLicense,
9          PublicKeyPem = publicKeyPem
10     };
11
12     return licenseCertificated;
13 }
14 private static readonly JsonSerializerOptions JsonSerializerOptionsEncode =
       new()
15 {
16     ReferenceHandler = ReferenceHandler.Preserve,
17     WriteIndented = false
18 };
```

```
19 private static byte[] SerializeLicense(LicenseContract license)
20 {
21     string licenseJson = JsonSerializer.Serialize(license,
           JsonSerializerOptionsEncode);
22     return Encoding.UTF8.GetBytes(licenseJson);
23 }
24 private static byte[] SignLicenseData(byte[] licenseBytes, string
       privateKeyPem)
25 {
26     using var rsa = RSA.Create();
27     rsa.ImportFromPem(privateKeyPem);
28     return rsa.SignData(licenseBytes, HashAlgorithmName.SHA256,
           RSASignaturePadding.Pkcs1);
29 }
30 private static string EncodeLicense(byte[] signature, byte[] licenseBytes)
31 {
32     using var ms = new MemoryStream();
33     int signatureOffset = 4;
34     ms.Write(BitConverter.GetBytes(signature.Length), 0, signatureOffset);
35     ms.Write(signature, 0, signature.Length);
36     ms.Write(licenseBytes, 0, licenseBytes.Length);
37     return Convert.ToBase64String(ms.ToArray());
38 }
```

La decodifica della licenza esegue i passaggi opposti (listing 3.19). Viene fatto la decodifica da Base64 per estrarre l'array di bytes della licenza unita alla firma. Le due vengono separate in base all'offset inserito in codifica tra la licenza e la firma. La firma viene poi controllata in base ai byte della licenza e alla chiave pubblica. Questo serve per evitare che il contenuto della licenza sia stato alterato. Siccome l'algoritmo Base64 è utilizzabile da tutti, è possibile esporre sia il contenuto della licenza che la firma; tuttavia proprio perchè la firma è stata ottenuta tramite chiave privata, l'alterazione della firma è molto complessa. Una volta validata la firma, il contenuto della licenza viene deserializzato e usato dal chiamante in GPOne.

Codice 3.19: Codice per la decodifica della licenza

```
1 public static LicenseContract? DecodeAndVerifyLicense(string base64License,
```

```
    string publicKeyPem)
{
    var fileBytes = DecodeBase64License(base64License);
    var signature = ExtractSignature(fileBytes);
    var licenseBytes = ExtractLicenseBytes(fileBytes);
    if (!VerifySignature(licenseBytes, signature, publicKeyPem))
        return null;
    return DeserializeLicense(licenseBytes);
}
private static readonly JsonSerializerOptions JsonSerializerOptionsDecode =
    new()
{
    ReferenceHandler = ReferenceHandler.Preserve
};
private static byte[] DecodeBase64License(string base64License)
{
    return Convert.FromBase64String(base64License);
}
private static byte[] ExtractSignature(byte[] fileBytes)
{
    int signatureLength = BitConverter.ToInt32(fileBytes, 0);
    int signatureOffset = 4;
    byte[] signature = new byte[signatureLength];
    Array.Copy(fileBytes, signatureOffset, signature, 0, signatureLength);
    return signature;
}
private static byte[] ExtractLicenseBytes(byte[] fileBytes)
{
    int signatureLength = BitConverter.ToInt32(fileBytes, 0);
    int signatureOffset = 4;
    int licenseDataOffset = signatureOffset + signatureLength;
    int licenseDataLength = fileBytes.Length - licenseDataOffset;

    byte[] licenseBytes = new byte[licenseDataLength];
    Array.Copy(fileBytes, licenseDataOffset, licenseBytes, 0,
        licenseDataLength);
```

```
35      return licenseBytes;
36  }
37  private static bool VerifySignature(byte[] licenseBytes, byte[] signature,
        string publicKeyPem)
38  {
39      using var rsa = RSA.Create();
40      rsa.ImportFromPem(publicKeyPem);
41      return rsa.VerifyData(licenseBytes, signature, HashAlgorithmName.SHA256,
            RSASignaturePadding.Pkcs1);
42  }
43  private static LicenseContract? DeserializeLicense(byte[] licenseBytes)
44  {
45      string licenseJson = Encoding.UTF8.GetString(licenseBytes);
46      return JsonSerializer.Deserialize<LicenseContract>(licenseJson,
            JsonSerializerOptionsDecode);
47  }
```

All'interno del servizio delle licenze, la chiamata di controllo effettuata da GPOne effettua dei ragionamenti sulla data di scadenza prima di richiamare i metodi della CryptoUtils. Se la licenza è stata bloccata, la licenza viene segnata come bloccata e la sua data di scadenza viene messa in un giorno del passato indipendentemente dalla sua data di scadenza; se la licenza è scaduta, la data di scadenza rimane inalterata; se la licenza è valida, la data di scadenza che verrà scritta nel file di licenza sarà al massimo sette giorni in più rispetto alla data in cui è stata chiamata la funzione di controllo. Questo per garantire che il controllo venga fatto regolarmente e, nel caso in cui il servizio non dovesse essere raggiungibile per qualche giorno, per non notificare subito sull'installazione del cliente che gli è scaduta la licenza.t

3.6 Gestione risorse Azure e CI/CD

Come detto in precedenza, l'applicazione delle licenze viene rilasciata su tre risorse di Azure che comunicano tra loro: App Services, SQL Server e Static Web App. Queste tre risorse sono poi presenti in due gruppi di risorse paralleli che formano i due ambienti di rilascio, uno di staging e test e l'altro di produzione, il tutto sotto un'unica sottoscrizione Azure (figura 2.11).

Le differenze tra i due ambienti si trovano principalmente nelle risorse di SQL Server. A livello di risorse computazionali, per il momento le risorse sono sullo stesso piano tariffario, il Basic B1, per App Services e SQL Server. Static Web App è stato istanziato con il piano tariffario gratuito, in quanto presenta già dei limiti abbastanza alti per quello che serve all'applicazione licenze. Tra i SQL Server di staging e produzione la differenza più grossa sta nella periodicità dei backups. Per il server di staging il backup totale del database è impostato a cadenza mensile e quello differenziale a cadenza bisettimanale. Per il server di produzione invece, il backup differenziale è impostato a cadenza giornaliera e quello totale ogni due giorni. Questo comporta che il numero di backups che vengono conservati è diverso tra le due risorse, in quanto serve uno storico maggiore sull'applicazione in ambiente di produzione.

Le risorse SQL Server in entrambi gli ambienti sono con un piano a consumo, quindi vengono accesi solo nel momento in cui arriva una richiesta. Questo è stato necessario in quanto la differenza di prezzo tra un utilizzo "serverless" e un utilizzo "provisioned" era troppo grande, specialmente per la mole di chiamate che sono previste anche per il futuro. Con il carico corrente delle risorse, la singola risorsa sarebbe costata ben oltre le dieci volte in più al mese e questo moltiplicato poi per una seconda risorsa. L'unica controindicazione in questo caso è il "cold start" della risorsa, ovvero il tempo di avvio iniziale che dev'essere tenuto in considerazione le prime volte che si fanno chiamate. Questo però per il carico attuale non è visto come un problema. Le risorse di App Services e Static Web App al contrario di SQL Server sono "provisioned", quindi sono sempre attive. In questo caso la differenza di costo era minima dunque si è preferito mantenere sempre attiva App Services (Static Web App non lascia nemmeno la scelta in questo caso). App Services e SQL Server sono stati istanziati nei server di Azure della zona "Italy North", mentre Static Web App viene gestita di standard su scala globale grazie alla CDN (Content Delivery Network) di Microsoft.

Su entrambi i gruppi di risorse sono stati messi avvisi automatici al raggiungimento di alcune soglie di spesa, tramite Azure Budgets. Non è stato impostato alcun limite di spesa, siccome il carico è talmente basso da stare sotto i 30 EUR mensili per entrambi i gruppi di risorsa per ora. Tuttavia per monitorare che non ci siano eventi imprevisti, gli avvisi sono stati messi rispettivamente a 10, 30 e 50 EUR mensili per i gruppi di risorse.

Per il monitoraggio delle prestazioni e dei log di App Services si è utilizzato la risorsa integrata di Azure Application Insights. Questa raccoglie tutti i dati, anche in tempo reale, dei consumi

di risorse di calcolo, come CPU e RAM, oltre che tutte le eccezioni e i codici di errore generati dall'applicazione. Dal portale di Azure, è possibile visualizzare dei grafici grazie ai dati raccolti, in figura 3.3 è stato riportato un esempio di grafico delle chiamate HTTP che hanno generato un errore 401, con periodo di riferimento che va da dal 1 Gennaio al 7 Febbraio 2026. Per il momento non è stata integrata la SDK di Application Insights all'interno delle soluzioni frontend e backend, tuttavia nelle iterazioni che verranno dell'applicazione questo sviluppo è già in programma.

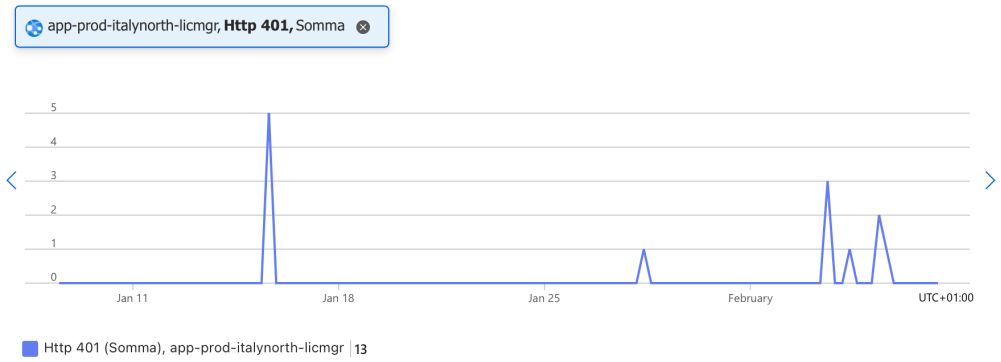

Figura 3.3: Grafico errori HTTP 401 su App Services

3.6.1 Pipelines di compilazione e rilascio

Il ciclo di CI/CD è stato gestito con l'utilizzo di due Azure Pipeline diverse. Una per la build, quindi per la compilazione delle soluzioni frontend e backend, e una per il deploy, il rilascio delle applicazione sulle risorse di Azure.

La pipeline di build viene usata ogni qualvolta è necessario fare il merge di una PR (Pull Request) sull'applicazione licenze. Per poter completare la PR è necessario che la compilazione vada a buon fine. Nel listing 3.20 si possono osservare tutti i passaggi che vengono fatti. I parametri iniziali riguardano il pool di agents disponibili per la compilazione e le immagini da usare; di default viene usato l'agent "backend" e l'immagine della versione più recente di Ubuntu. La pipeline ha due trigger, quindi viene lanciata in due casi:

- quando c'è una qualsiasi operazione riguardante il ramo develop, che fa da ponte tra il ramo master di Git e i rami degli sviluppi in corso, dunque sia quando siamo in una PR che deve andare su develop, sia nel momento successivo in cui viene effettuato il merge su develop;

- quando viene staccato un tag che inizia per "vprod".

Il tag è un modo per contrassegnare l'ultimo commit fatto su un ramo fino a quel momento. Questo serve per tenere traccia del punto in cui si è deciso di staccare una versione, oppure di tenere traccia dell'ultima volta che è stato mandato il codice sulle risorse di produzione. Il job della pipeline di build è uno solo, ed è quello di compilare sia il frontend che il backend. Per farlo è necessario installare Dotnet Tool, NPM e Node.js. Questo va fatto sempre perchè l'agent lavora su un'immagine per eseguire la compilazione e una volta effettuata la compilazione, l'agent riparte sempre da zero. Per quanto riguarda il backend, vengono fatti il build della soluzione e il lancio di uno strumento di test, che esegue tutti i test contenuti in un progetto composto da unit tests. Solo per il backend è necessario poi eseguire il comando publish di dotnet e il comando di archiviazione dei file generati, che andrà a generare un archivio nella cartella di compilazione dell'agent. Per il frontend in questo caso è necessario solo la compilazione.

Codice 3.20: Pipeline di build applicazione licenze

```
1  parameters:
2    - name: poolName
3      displayName: Pool Name
4      default: Backend
5      values:
6        - Backend
7        - Default
8        - Azure Pipelines
9    - name: poolImage
10     displayName: Pool Image
11     default: ubuntu-latest
12     values:
13       - ubuntu-latest
14       - windows-latest
15
16 trigger:
17   branches:
```

```
18      include:
19        - develop
20    tags:
21      include:
22        - vprod*
23
24  jobs:
25    - job: build
26      displayName: Build and Test
27      workspace:
28        clean: all
29      pool:
30        name: ${{ parameters.poolName }}
31        vmImage: ${{ parameters.poolImage }}
32
33      steps:
34        - checkout: self
35          clean: true
36          lfs: false
37
38        - script: dotnet tool restore --ignore-failed-sources
39          displayName: Dotnet tool restore
40
41        - task: NuGetToolInstaller@1
42        - task: NuGetAuthenticate@1
43
44        - script: dotnet restore $(Build.SourcesDirectory)\Genba.
      CloudLicenses
45          displayName: Restore .net dependencies all
46
47        - script: dotnet build $(Build.SourcesDirectory)\Genba.
      CloudLicenses --configuration $(BuildConfiguration)
```

```
48        displayName: Build backend
49
50      - task: DotNetCoreCLI@2
51        displayName: Test
52        inputs:
53          command: 'test'
54          projects: '$(Build.SourcesDirectory)\Genba.CloudLicenses'
55          arguments: '--configuration Release --filter EXECUTION!=SLOW'
56
57      - task: NodeTool@0
58        displayName: Install nodejs
59        inputs:
60          versionSpec: 22.x
61
62      - task: Npm@1
63        inputs:
64          command: 'install'
65
66      - script: npm run build --c release
67        displayName: Build release
68
69      - script: dotnet publish $(Build.SourcesDirectory)/Genba.
    CloudLicenses/Genba.CloudLicenses.Api --configuration $(
    BuildConfiguration) --runtime win-x64 --output $(Build.
    ArtifactStagingDirectory)/api
70        displayName: Publish api
71
72      - task: ArchiveFiles@2
73        displayName: Archive Genba.CloudLicenses.Api
74        inputs:
75          rootFolderOrFile: '$(Build.ArtifactStagingDirectory)/api'
76          includeRootFolder: false
```

```
77          archiveType: 'zip'
78          archiveFile: '$(Build.ArtifactStagingDirectory)/zip/Genba.
    CloudLicenses.Api.zip'
79          replaceExistingArchive: true
```

Una volta che viene effettuata la compilazione del ramo develop o del ramo contrassegnato con il tag, viene lanciata la pipeline di rilascio. Nel listing 3.21 è stato riportato solo lo stage relativo all'ambiente staging. La pipeline completa avrebbe un secondo stage per l'ambiente di produzione, che risulta sostanzialmente identico allo stage di staging con la sola differenza delle variabili che vengono utilizzate. In questo caso è presente un trigger collegato alla pipeline di compilazione, ogni volta che la compilazione del ramo develop o di un ramo con tag "vprod" va a buon fine. Nella pipeline di rilascio inoltre è presente anche un gruppo di variabili che viene utilizzato. Questo serve per non mettere in chiaro nel file YAML alcuni parametri che vanno passati ai tasks della pipeline, ad esempio l'id della sottoscrizione Azure. Lo stage di staging ha una condizione di attivazione, ovvero che il branch deve essere il branch di develop e che il tag non inizi per "vprod". La condizione opposta è usata nello stage di produzione. Il job è uno solo in questo caso e si occupa sia di rilasciare la Static Web App che l'App Services. Si è deciso per il momento di metterli nello stesso job in quanto non è un'operazione troppo onerosa dal punto di vista temporale; per la stessa motivazione, si è scelto di avere un'unica pipeline di compilazione per frontend e backend. Per entrambi i tasks di rilascio, Azure ha messo a disposizione dei task standard richiamabili in pipeline che gestiscono tutto il rilascio sulle risorse Azure. Il primo task riguarda Static Web App, e richiede solo i percorsi dell'applicazione frontend e della sua cartella di compilazione, oltre al token (univoco) di distribuzione della risorsa su Azure, che è stato memorizzato nella libreria delle variabili "genba-cloud-licenses-deploy"; il secondo task esegue il medesimo compito per App Services, in questo caso prendedo in input l'id della sottoscrizione Azure, il nome della web app e il nome del gruppo di risorse, oltre al percorso dell'artefatto generato nella pipeline precedente.

Codice 3.21: Pipeline di deploy applicazione licenze

```
1 trigger: none
2
3 resources:
4   pipelines:
```

```
5     - pipeline: CI_CloudLicenses
6       source: genba-cloudlicenses-build
7       trigger:
8         branches:
9           include:
10            - develop
11            - refs/tags/vprod*
12
13 variables:
14   - group: genba-cloud-licenses-deploy
15
16 stages:
17   - stage: CD_Staging
18     displayName: CD Staging
19     condition: and(eq(variables['Build.SourceBranch'], 'refs/heads/
     develop'), not(startsWith(variables['Build.SourceBranch'], 'refs/
     tags/vprod')))
20     jobs:
21       - deployment: Staging
22         displayName: Staging deploy
23         environment:
24           name: Staging
25           resourceType: VirtualMachine
26           tags: staging
27         workspace:
28           clean: all
29         strategy:
30           runOnce:
31             deploy:
32               steps:
33                 - task: AzureStaticWebApp@0
34                   inputs:
```

```
35                  app_location: '/Genba.CloudLicenses.Frontend/
    CloudLicenses'
36                  output_location: '/Genba.CloudLicenses.Frontend/
    CloudLicenses/dist/browser'
37                  azure_static_web_apps_api_token: '$(
    AzureSWATokenStaging)''
38              - task: AzureWebApp@1
39                inputs:
40                  azureSubscription: '$(AzureSubscriptionName)'
41                  appType: 'webApp'
42                  appName: '$(StagingWebAppName)'
43                  resourceGroupName: '$(StagingResourceGroup)'
44                  package: '$(Pipeline.Workspace)\CI_Backend\drop\$(
    PackageName)*.zip'
45                  deploymentMethod: 'auto'
```

3.7 Implementazioni su GPOne

Lo sviluppo dell'applicazione delle licenze è stato portato avanti in parallelo alle modifiche sulla soluzione di GPOne. Le modifiche sono state principalmente sulla parte backend. L'obiettivo era ottenere una struttura come quella in figura 3.4, ovvero aggiungere un microservizio che facesse da tramite tra GPOne e l'applicazione licenze, sia in autonomia con una chiamata periodica, sia a seguito di una richiesta di "aggiornamento licenze" proveniente dal client MES di GPOne. Il nuovo microservizio avrebbe poi comunicato in broadcast a tutti i servizi che utilizzavano la licenza dell'avvenuta modifica e ognuno di essi avrebbe infine aggiornato le informazioni nella propria memoria della licenza.

3.7.1 Aggiunta microservizio di controllo licenza

Il microservizio implementato è abbastanza leggero. Il servizio al suo interno instanzia due oggetti:

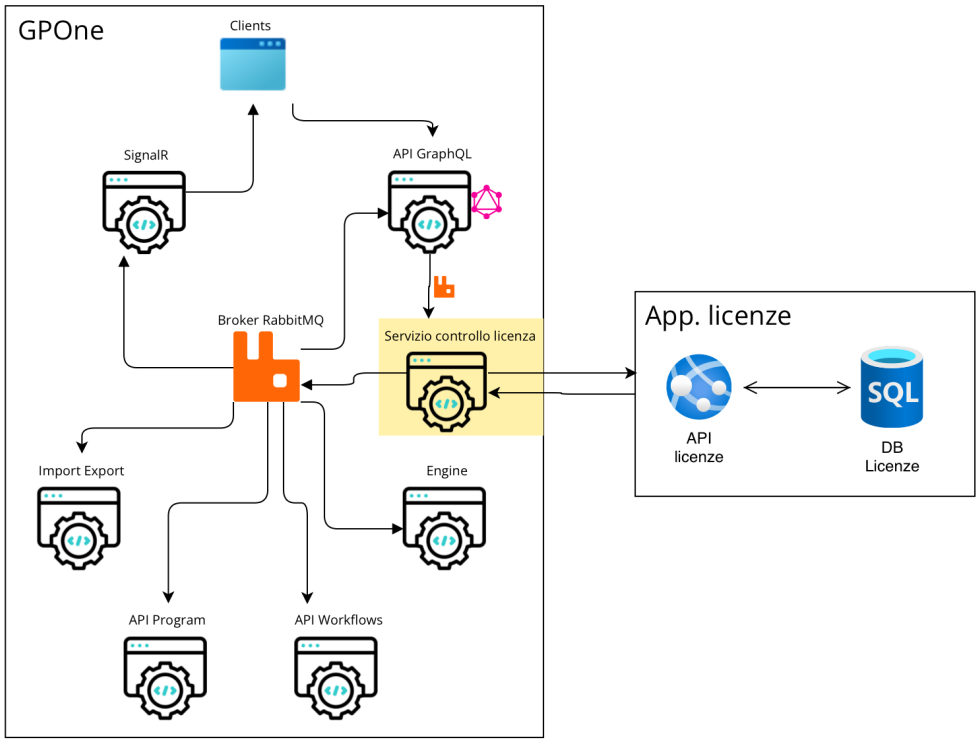

Figura 3.4: Architettura complessiva (semplificata) GPOne - App.licenze

- un consumer di MassTransit per RabbitMQ, per le richieste che arrivano dal servizio API GraphQL da parte dei client MES;
- un job periodico, a orario personalizzabile, che una volta al giorno esegua un aggiornamento del file di licenza.

La maggior parte del lavoro viene svolto dalla libreria di MassTransit. Nel listing 3.22 viene registrato in dependency injection tutto l'occorrente per l'utilizzo del bus di MassTransit. All'interno della configurazione, è possibile registrare i consumer con il metodo AddConsumer ed è possibile dichiarare il sistema di messaggistica sottostante, in questo caso si è dichiarato l'utilizzo di RabbitMQ. Chiaramente è necessario che il broker di RabbitMQ sia installato sulla macchina di GPOne oppure che sia raggiungibile nella stessa rete di GPOne. La flessibilità di MassTransit è data proprio da questa capacità di poter dichiarare con un semplice metodo cosa si vuole usare come sistema di messaggistica senza andare a cambiare il codice degli eventi e dei consumers.

Codice 3.22: Dependency injection di MassTransit e RabbitMQ

```
builder.Services.AddMassTransit(serviceCollectionBusConfigurator =>
{
    serviceCollectionBusConfigurator.AddConsumer(typeof(
        CheckLicenseEventConsumer), typeof(CheckLicenseEventConsumerDefinition)
        );
    serviceCollectionBusConfigurator.SetKebabCaseEndpointNameFormatter();
    serviceCollectionBusConfigurator.UsingRabbitMq((context,
        rabbitMqBusFactoryConfigurator) =>
    {
        var configSection = configuration.GetSection("RabbitMq");
        var host = configSection["Host"];
        var virtualHost = configSection["VirtualHost"];
        var port = Convert.ToUInt16(configSection["Port"]);
        var userName = configSection["Username"];
        var password = configSection["Password"];
        rabbitMqBusFactoryConfigurator.Host(host,
            port,
            virtualHost,
            h =>
            {
                h.Password(password);
                h.Username(userName);
            });
        rabbitMqBusFactoryConfigurator.ConfigureEndpoints(context);
    });
});
```

L'implementazione del consumer (listing 3.23) avviene creando una classe che estenda l'interfaccia IConsumer di MassTransit. Nel caso di GPOne, si è scelto anche di implementare per i consumers una classe che estenda la classe astratta di ConsumerDefinition, in modo da poter dare un nome specifico all'endpoint del consumer. Questo è stato necessario in passato perchè erano stati implementati due consumers che consumavano lo stesso evento, e non avendo distinto gli endpoint ma avendo solo registrato due consumers, i due si dividevano i messaggi in modo round-robin, anche se il comportamento che si voleva ottenere era di una comunicazione a entrambi i consumers del messaggio. Da allora, si è mantenuta questa convenzione per tutti i consumers di

GPOne. Anche per i consumers dell'evento di licenza aggiornata sui diversi microservizi sono stati usati nomi di endpoint diversi; in questo modo la comunicazione dell'evento di licenza aggiornata avviene effettivamente in modalità broadcast.

Codice 3.23: Consumer messaggio RabbitMQ con MassTransit

```
1  using Genba.BackEnd.Domain.Common;
2  using Genba.BackEnd.Domain.License.Events;
3  using Genba.CloudLicenseChecker.Services;
4  using MassTransit;
5  using Microsoft.Extensions.Logging;
6  namespace Genba.CloudLicenseChecker.Consumers.CheckLicense;
7  public class CheckLicenseEventConsumer(
8      ILogger<CheckLicenseEventConsumer> logger,
9      ILicenseRewriter licenseRewriter) : IConsumer<ICheckLicenseEvent>
10 {
11     private readonly ILogger<CheckLicenseEventConsumer> _logger = logger;
12     private readonly ILicenseRewriter _licenseRewriter = licenseRewriter;
13     public Task Consume(ConsumeContext<ICheckLicenseEvent> context)
14     {
15         try
16         {
17             _licenseRewriter.Execute();
18         }
19         catch (Exception ex)
20         {
21             _logger.LogError(ex.Message);
22         }
23         return Task.CompletedTask;
24     }
25 }
26
27 public class CheckLicenseEventConsumerDefinition : ConsumerDefinition<
       CheckLicenseEventConsumer>
28 {
29     public CheckLicenseEventConsumerDefinition()
30     {
```

```
31          EndpointName = QueuesNames.CheckLicenseQueueName;
32      }
33  }
```

L'implementazione dell'evento è abbastanza blanda (listing 3.24). L'evento utilizza un attributo EntityName che dichiara il nome dell'evento all'interno di MassTransit e di conseguenza di RabbitMQ, in modo da poterlo filtrare nella console di controllo di RabbitMQ. Nell'evento viene poi utilizzato l'oggetto IBusControl di MassTransit, che è l'astrazione con cui vengono spediti e gestiti i messaggi. All'interno di GPOne gli unici due modi che vengono usati sono il Publish, ovvero l'invio del messaggio in modalità "fire and forget", dunque senza attendere una risposta dal consumer, e la Request, in questo caso attendendo la risposta del consumer e nel caso non arrivi entro il tempo di timeout dando errore.

Codice 3.24: Evento di controllo licenza

```
1  using MassTransit;
2  namespace Genba.BackEnd.Domain.License.Events;
3  [EntityName("CHECK-LICENSE")]
4  public interface ICheckLicenseEvent
5  {
6  }
7  public class CheckLicenseEvent : ICheckLicenseEvent
8  {
9      public static void Send(IBusControl bus)
10     {
11         bus.Publish<ICheckLicenseEvent>(
12             new CheckLicenseEvent(),
13             publishContext =>
14             {
15                 publishContext.Durable = false;
16             });
17     }
18 }
```

Il cuore del microservizio è il servizio di riscrittura licenza (listing 3.25). La classe espone un unico metodo pubblico che esegue il compito in tre step:

- viene controllato che la licenza attuale non sia stata spostata o alterata, con il metodo GetLicense dell'oggetto LicenseUtil, contenuto nel pacchetto Genba.License e contiene una serie di funzioni condivise per la lettura della licenza;
- viene chiamata l'API dell'applicazione delle licenze a un endpoint specifico che prende la chiave di attivazione associata alla licenza e l'identificativo del server a 16 caratteri da cui è stata fatta la chiamata, utilizzando una funzione contenuta nel pacchetto Genba.License, la stessa che veniva usata per controllare la validità della licenza prima delle implementazioni viste;
- una volta ottenuta la licenza, viene riscritto il file fisico di licenza sul server e viene mandato un evento di licenza aggiornata con MassTransit a tutti i consumers registrati nei microservizi in ascolto su GPOne.

Nella riscrittura del file di licenza viene comunque usato un ulteriore livello di codifica basandosi sull'indirizzo MAC della prima scheda di rete del server. Questo è un passaggio necessario altrimenti la licenza sarebbe trasferibile su un'altra installazione senza problemi, nel caso di installazioni offline in particolare. Si è scelto di utilizzare l'indirizzo MAC in quanto questo tende a variare molto meno sulle macchine virtuali dei clienti che si basano solitamente su un unico server all'interno del perimetro aziendale. Il caso di migrazioni delle istanze su server completamente diversi è molto raro tra i client per questo in quei casi rimane il problema del dover riaggiornare la licenza, ma per il caso base non dovrebbe creare problemi.

Codice 3.25: Servizio di controllo licenza

```
1  using Genba.BackEnd.Common.ActionUtils;
2  using Genba.BackEnd.Common.Configurations;
3  using Genba.BackEnd.Domain.License.Events;
4  using Genba.License.CryptographyUtils;
5  using Genba.License.LicenseUtils;
6  using Genba.License.SharedModels;
7  using MassTransit;
8  using Microsoft.Extensions.Configuration;
9  using Microsoft.Extensions.Logging;
10 using Newtonsoft.Json;
```

```
11 namespace Genba.CloudLicenseChecker.Services;
12 public interface ILicenseRewriter
13 {
14     void Execute();
15 }
16 public class LicenseRewriter(
17     IHttpClientFactory httpClientFactory,
18     ILogger<LicenseChecker> logger,
19     ILicenseUtil licenseUtil,
20     IBusControl busControl) : ILicenseRewriter
21 {
22     private readonly HttpClient _httpClient = httpClientFactory.CreateClient()
           ;
23     private readonly ILogger<LicenseChecker> _logger = logger;
24     private readonly IConfiguration configuration = ConfigurationFactory.
           Create();
25     private readonly ILicenseUtil _licenseUtil = licenseUtil;
26     private readonly IBusControl _busControl = busControl;
27
28     public async void Execute()
29     {
30         try
31         {
32             LicenseContract oldLicenseContract = null;
33             _licenseUtil.GetLicense(license => oldLicenseContract = license,
                   ActionNull.Execute);
34             if(oldLicenseContract == null)
35             {
36                 _logger.LogError("Error getting current license");
37                 return;
38             }
39             var newLicense = await GetNewLicense(oldLicenseContract.ProductKey
                   );
40             WriteNewLicense(newLicense);
41             UpdatedLicenseEvent.Send(_busControl);
42         }
```

```
43          catch (Exception ex)
44          {
45              _logger.LogError(ex.Message);
46          }
47      }
48      private async Task<string> GetNewLicense(string productKey)
49      {
50          var checkUrl = configuration.GetSection("CheckApiCallUrl").Value;
51          string url = $"{checkUrl}?productKey={Uri.EscapeDataString(productKey)
                }&deviceId={Uri.EscapeDataString(CryptoUtils.GetDeviceIdentifier())
                }";
52          var licenseResponse = await _httpClient.GetStringAsync(url);
53          if (licenseResponse != null && string.IsNullOrWhiteSpace(
                licenseResponse))
54          {
55              _logger.LogError("Cloud license manager did not return any license
                    ");
56              throw new Exception("Cloud license manager did not return any
                    license");
57          }
58          var licenseCertificated = JsonConvert.DeserializeObject<
                LicenseCertificated>(licenseResponse!);
59          if (licenseCertificated == null)
60          {
61              _logger.LogError("Error parsing license certificated");
62              throw new Exception("Error parsing license certificated");
63          }
64          var licenseDecoded = CryptoUtils.DecodeAndVerifyLicense(
                licenseCertificated.LicenseEncoded, licenseCertificated.
                PublicKeyPem);
65          if (licenseDecoded == null)
66          {
67              _logger.LogError("Error decoding license");
68              throw new Exception("Error decoding license");
69          }
70          return licenseResponse;
```

```
71      }
72      private void WriteNewLicense(string newLicense)
73      {
74          var licensePath = configuration.GetSection("LicensePath").Value;
75          var directoryPath = Path.GetDirectoryName(Path.GetFullPath(licensePath
                ));
76          var oldLicenseCopyPath = Path.Combine(directoryPath, "old_license");
77          File.Copy(licensePath, oldLicenseCopyPath);
78          try
79          {
80              var macEncodedLicense = CryptoUtils.CreateMacEncodedFile(
                    newLicense);
81              File.WriteAllText(licensePath, macEncodedLicense);
82          }
83          catch (Exception ex)
84          {
85              _logger.LogError("Error recreating license");
86              File.Move(oldLicenseCopyPath, licensePath);
87              throw new Exception("Error recreating license");
88          }
89          File.Delete(oldLicenseCopyPath);
90          return;
91      }
92  }
```

3.7.2 Modifiche sui microservizi esistenti e sul setup

I punti da modificare sui servizi preesistenti non erano tanti, tuttavia complice anche anni di scrittura non ottimale di codice manutenibile, si sono dovuti adeguare anche un sacco di classi che chiamavano alcune funzioni condivise. Il numero di files che sono stati modificati tra le varie soluzioni di GPOne sono stati oltre 300.

Sulle soluzioni dei clients è stato necessario solo riadeguare i punti in cui veniva fatto il controllo dei moduli attivi, che chiedeva sia un'applicazione che un modulo, facendo richiedere solo il modulo, vista la normalizzazione a singolo livello fatta sul file di licenza, e le mutations di GraphQL che

gestivano il login. Le modifiche più sostanziali sono state fatte sul backend. Siccome non c'era uniformità di gestione dei controlli dei moduli e per avere un punto unico di controllo licenza per ogni singolo servizio, è stato creata una classe che esponesse come metodi pubblici tutte le varie funzioni per controllare moduli e permessi. Nel listing 3.26 si può osservare una parte della classe in questione, LicenseManager. La classe viene registrata in dependency injection come servizio singleton, e una volta istanziata va a leggere il file fisico della licenza e va a caricare i permessi per la visibilità di griglie e form e per eseguire operazioni di CRUD. I permessi abilitati per ogni modulo sono salvati in dei file JSON e LicenseManager all'avvio si salva nella cache tutti i permessi abilitati (all'interno di IMemoryCache della libreria di Microsoft). I metodi che la classe espone sono:

- una chiamata di lettura della licenza, salvata anch'essa in memoria che ritorna il contenuto del file della licenza;
- una chiamata per ottenere i codici di tutti i moduli attivi;
- una chiamata per ottenere i permessi attivi in quel momento per i moduli;
- una chiamata per controllare che la licenza non sia scaduta e una per la scadenza del singolo modulo;
- una chiamata per controllare il numero di abilitazioni attive per alcuni moduli;
- una chiamata per i consumers che consente di rileggere il file fisico della licenza a runtime.

Codice 3.26: Classe singleton di gestione licenza su GPOne

```
1 using Genba.BackEnd.Common.ActionUtils;
2 using Genba.BackEnd.Common.Assets;
3 using Genba.BackEnd.Model.License.Modules;
4 using Genba.BackEnd.Model.Shell.Grid;
5 using Genba.BackEnd.UtilityKit.Consts;
6 using Genba.License.LicenseUtils;
7 using Genba.License.SharedModels;
8 using Microsoft.Extensions.Caching.Memory;
9 using Microsoft.Extensions.Configuration;
```

```
10 using Microsoft.Extensions.Logging;
11 using System;
12 using System.Collections.Generic;
13 using System.IO;
14 using System.Linq;
15 using System.Text.Json;
16 using Actions = Genba.BackEnd.Model.Shell.Actions;
17 namespace Genba.BackEnd.Model.Common.Security.Configuration;
18 public class LicenseManager : ILicenseManager
19 {
20     private List<Actions.Action> _availableActions;
21     private LicenseContract _license;
22     private List<ModuleContract> _modules;
23     private IAssetReader _assetReader;
24     private ILicenseUtil _licenseUtil;
25     private readonly IMemoryCache _memoryCache;
26     private readonly ILogger<SecurityConfiguration> _logger;
27     public LicenseManager(ILicenseUtil licenseUtil,
28                           IAssetReader assetReader,
29                           IConfiguration configuration,
30                           IMemoryCache memoryCache,
31                           ILogger<SecurityConfiguration> logger)
32     {
33         _assetReader = assetReader;
34         _logger = logger;
35         _licenseUtil = licenseUtil;
36         ReadUpdateLicense();
37 
38         _memoryCache = memoryCache;
39         LoadAvailableActions(configuration.GetSection("AssetsFolder").Value);
40     }
41     public LicenseContract GetLicense()
42     {
43         return _license;
44     }
45     public int GetModuleNumberOfEnabledUsers(string moduleName)
```

```
46      {
47          return _license.LicenseModules.FirstOrDefault(m => m.TechnicalCode ==
                moduleName)?.NumberOfLicenses ?? 0;
48      }
49      public bool IsLicenseExpired()
50      {
51          return _license.CheckExpireDate.HasValue ? _license.CheckExpireDate.
                Value < DateTime.UtcNow : false;
52      }
53      public bool ReadUpdateLicense()
54      {
55          var isError = false;
56
57          _licenseUtil.GetLicense(
58              updatedLicense =>
59              {
60                  _license = updatedLicense;
61              },
62              () =>
63              {
64                  isError = true;
65              });
66
67          return isError;
68      }
69      public List<Actions.Action> GetAvailableActionsInActiveModules()
70      {
71          var key = $"AvailableActionsInActiveModules";
72          var cachedAvailableActionInActiveModule = (List<Actions.Action>)
                _memoryCache.Get(key);
73          if (cachedAvailableActionInActiveModule == null)
74          {
75              cachedAvailableActionInActiveModule = _memoryCache.GetOrCreate(
76              key,
77              entry =>
78              {
```

```
79                 entry.SlidingExpiration = TimeSpan.FromSeconds(20);
80                 var activeModules = GetModulesForLicense();
81                 var availableActionInActiveModule = new List<Actions.Action>()
                       ;
82                 foreach (var activeModule in activeModules)
83                 {
84                     availableActionInActiveModule.AddRange(
85                         _availableActions.Where(action => action.ModulesCodes.
                               Contains(activeModule.TechnicalCode)
86                         && availableActionInActiveModule.Any(p => p.Name ==
                               action.Name) == false));
87                 }
88                 return availableActionInActiveModule;
89             });
90         }
91         return cachedAvailableActionInActiveModule;
92     }
93     public bool IsModuleActive(string moduleName)
94     {
95         return _license.LicenseModules.Any(
96             m => m.TechnicalCode == moduleName && (m.ExpireDate.HasValue ==
                   false || m.ExpireDate > DateTime.UtcNow));
97     }
98     public List<ModuleContract> GetActiveModules()
99     {
100        return _license.LicenseModules.Where(m => m.ExpireDate.HasValue ==
               false || m.ExpireDate > DateTime.UtcNow).ToList();
101    }
102 }
```

Per i servizi API GraphQL, API Workflows, API Program, Engine, Import Export è stato istanziato un consumer per l'evento di aggiornamento licenza. Questo consumer andrà poi a richiamare il metodo ReadUpdateLicense della classe LicenseManager.

Altre modifiche riguardano la gestione del numero di abilitazione per i moduli Base, MachineLink, Fdc, Program. La logica dietro ognuno di questi è che non dev'essere possibile abilitare un numero maggiore di utenti o dispositivi previsti per il modulo. Per MachineLink, che riguarda

il numero di collegamenti driver che vengono attivati nell'Engine per le macchine utensili, Fdc e Program, che entrambi pilotano il numero di dispositivi Program e Fdc abilitati, è stato solo corretto il codice rispetto alle modifiche, ma la logica è rimasta come prima siccome era già corretta. Per la gestione degli utenti del modulo Base invece è stata modificata la logica per funzionare come gli altri moduli. L'idea iniziale era di consentire un numero massimo in contemporanea di utenti attivi nel sistema, tuttavia si è visto col tempo che questa gestione provocava delle difficoltà sia in assistenza che per gli sviluppatori. La chiamata per il login è stata semplificata e non considera più il numero di utenti attivi in contemporanea, tracciato in base al numero di token salvati a database; un utente adesso può essere o meno abilitato indipendentemente dal numero di connessioni al sistema. Per gestire il numero massimo di abilitazioni sono stati modificati i validatori di creazione e aggiornamento degli utenti, per controllare al tentativo di abilitazione di un utente non si ecceda la soglia prevista dal modulo.

L'ultima modifica effettuata è stata sul programma di installazione di GPOne. All'interno delle prime schermate del setup è stata aggiunta la possibilità di passare la chiave di attivazione della licenza di GPOne oltre alla licenza fisica (figura 3.5). Inoltre è stato aggiunto all'interno del processo l'installazione del servizio di controllo licenza. È prevista un'opzione che non attivi il servizio, in quanto sulle reti chiuse il servizio ritornerebbe sempre un errore e alla lunga invaliderebbe la licenza. Nel setup viene fatto ovviamente lo stesso passaggio di codifica della licenza con l'indirizzo MAC della macchina.

3.8 Test e benchmarks

L'ultima parte dello sviluppo prima di interfacciarsi con le installazioni cliente ha riguardato la parte di test e di benchmarks di alcuni punti dell'applicazione.

3.8.1 Test effettuati

IL reparto qualità si è occupato di testare i flussi nella loro interezza. Il compito del reparto si è diviso in tre parti:

- test dell'applicazione delle licenze, delle interazioni utente e delle chiamate API;

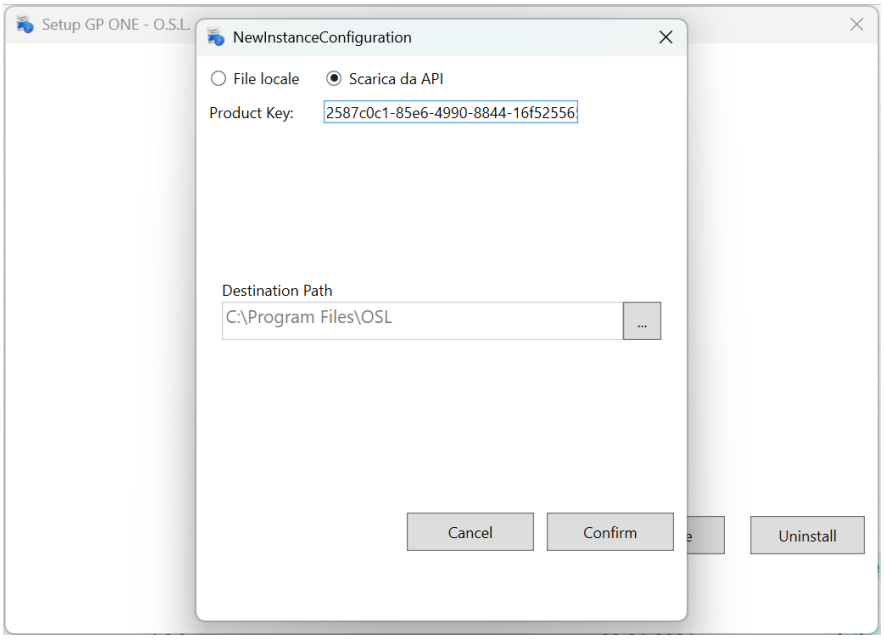

Figura 3.5: Step di installazione di richiesta chiave di attivazione

- test della versione aggiornata di GPOne;
- test sui flussi di lavoro, dall'inserimento dell'ordine cliente all'installazione effettiva.

Il test dell'applicazione licenze ha riportato principalmente dei feedbacks e suggerimenti lato UI, come la modifica del posizionamento di alcuni pulsanti, e alcuni bug semplici relativi all'inserimento di nuove versioni di licenze e alle etichette di traduzione sul client.

Il test di GPOne è stato quello che ha riportato i problemi maggiori, in quanto è andato a cambiare pesantemente la parte di login dell'applicazione e di abilitazione degli utenti. I problemi sono stati principalmente legati al numero di utenti precedentemente attivi su una licenza, che era più basso rispetto agli utenti effettivamente abilitati sulla licenza. Questo andava a bloccare l'accesso al sistema. Queste segnalazioni hanno portato a una serie di aggiustamenti sia nella logica di accesso sia nei validatori delle entità che avevano il numero di abilitazioni legato a un modulo della licenza.

Per quanto riguarda i test dei flussi, il flusso di inserimento a partire dall'inserimento dell'ordine cliente non ha mostrato alcun problema. Nel flusso di installazione invece c'è stato un unico problema nel caso di aggiornamento di un'installazione preesistente. Il problema era legato alla

ricerca del vecchio file di licenza da andare a sostituire, tuttavia il programma d'installazione si aspettava già un file nel formato della nuova licenza. Essendo il numero di clienti molto limitato, questo errore non è stato risolto lato codice ma aggirato copiando all'interno della cartella d'installazione un file nel formato nuovo della licenza.

3.8.2 Benchmarks effettuati

La parte di benchmarks non è stata particolarmente articolata. I benchmarks effettuati sono stati eseguiti principalmente su alcune chiamate dell'API e su alcune query sul database di SQL Server. Non essendo un'applicazione troppo complessa o con un alto numero di componenti frontend, non è stato svolto alcun benchmark sulle prestazioni della pagina web, in quanto i colli di bottiglia allo stato attuale eventualmente sarebbero stati solo nella parte backend. Si analizzeranno due casi tra i test effettuati.

Per SQL Server, sono state eseguite principalmente queries sulle tabelle Licenses e License-Modules. Questo perchè sono le due tabelle che verranno interrogate più spesso, sia da remoto per il controllo della licenza di GPOne, sia per i flussi aziendali interni. L'obiettivo dei test effettuati su SQL è stato monitorarne i tempi di risposta, la percentuale di CPU utilizzata dalla risorsa e il numero di DTU. I tempi di risposta danno un'indicazione sul fatto che gli indici sulla tabella siano già sufficienti o se ne servono degli altri. Il monitoraggio di CPU e DTU serve invece per controllare che il livello Basic B1 della risorsa SQL Server sia sufficiente anche in caso di un carico più alto del normale.

La query utilizzata nel listing 3.27 è una delle combinazioni di query eseguita più spesso in quanto va a simulare le chiamate che vengono fatte dall'API quando un'installazione di GPOne richiede un controllo licenza, passando una chiave di attivazione come parametro. I clienti attuali sommati di tutti i prodotti aziendali è intorno a 3000; anche convertendo tutti i clienti attuali a usare GPOne, non si prevede che si andrà molto oltre. Si è comunque deciso di metterlo sotto stress con un caso un po' più estremo, impostando il numero di chiamate SQL in sequenza a 5000.

Codice 3.27: Esempio di query test eseguita sul database delle licenze

```
1 SET STATISTICS TIME ON;
2 SET STATISTICS IO ON;
3 DECLARE @Counter INT = 0;
```

```
DECLARE @StartTime DATETIME = GETDATE();
WHILE @Counter < 5000
BEGIN
    SELECT TOP 1 * FROM [dbo].[Licenses] where ProductKey = 'f16f5182-5faa-418
       a-9d80-8aae15d3e7cf' and DeletedDateTime is null;
    SELECT lm.* FROM [dbo].[LicenseModules] lm join [dbo].[Licenses] l on l.Id
        = lm.LicenseId where ProductKey = 'f16f5182-5faa-418a-9d80-8
       aae15d3e7cf' and DeletedDateTime is null;
    SET @Counter = @Counter + 1;
END
SELECT DATEDIFF(ms, @StartTime, GETDATE()) AS OverallQueryExecutionTime;
SET STATISTICS TIME OFF;
SET STATISTICS IO OFF;
```

La query è stata effettuata con circa 3000 record di licenze fittizie con ognuna 4 moduli attivi, dunque 12000 record all'interno di LicenseModules, ed è stata eseguita 100 volte. I risultati hanno mostrato un buon comportamento della risorsa. Il tempo medio di attesa dello script T-SQL è stato di 4200 millisecondi mentre la percentuale media di CPU utilizzata è stata 74%, con un numero di DTU medio pari a 3.60. Essendo un caso limite, anche se si avvicina ai limiti di risorse, fintanto che il numero di clienti non aumenterà si manterrà il Basic B1 come livello di Azure SQL Server.

Per quanto riguarda l'API, la chiamata di controllo licenza remoto è stata testata utilizzando uno script AngularTS di K6, un tool per il test delle prestazioni realizzato da Grafana. Il listing 3.28 mostra uno script realizzato per testare l'endpoint della chiamata di controllo licenza. Le impostazioni nella struttura options impostano la durata del test a 180 secondi, con un numero di utenti virtuali contemporanei (vus) a 100. Inoltre la funzione check permette di riportare immediatamente se il test è stato superato o meno in base alle metriche che si scelgono di controllare, in questo caso che le risposte siano state tutte con codice 200 e che il tempo di risposta sia stato minore di 500 millisecondi.

```
import http from 'k6/http';
import { sleep, check } from 'k6';
export let options = {
  vus: 100,
  duration: '180s',
};
```

```
7  export default function () {
8    let res = http.get('https://app-prod-italynorth-licmgr.azurewebsites.net/api
       /licenses/check?productKey=prodkey&deviceId=deviceid');
9    check(res, {
10     'Response 200': (r) => r.status === 200,
11     'Resp. time < 500ms': (r) => r.timings.duration < 500,
12   });
13   sleep(1);
14 }
```

Codice 3.28: Esempio di script k6

Il test in questo caso ha mostrato un aumento dei tempi di risposta all'aumentare del numero di richieste. La media è stata di 703 millisecondi, con un il tempo di risposta minimo di 21,69 millisecondi e il massimo di 2,92 secondi, a fronte di oltre 10000 richieste effettuate nell'arco di 3 minuti da 100 utenti diversi. Nonostante ciò, le chiamate hanno riportato tutte un esito positivo; inoltre, analizzando il database, durante il test non ha mai utilizzato oltre il 30% di CPU e oltre 1 DTU.

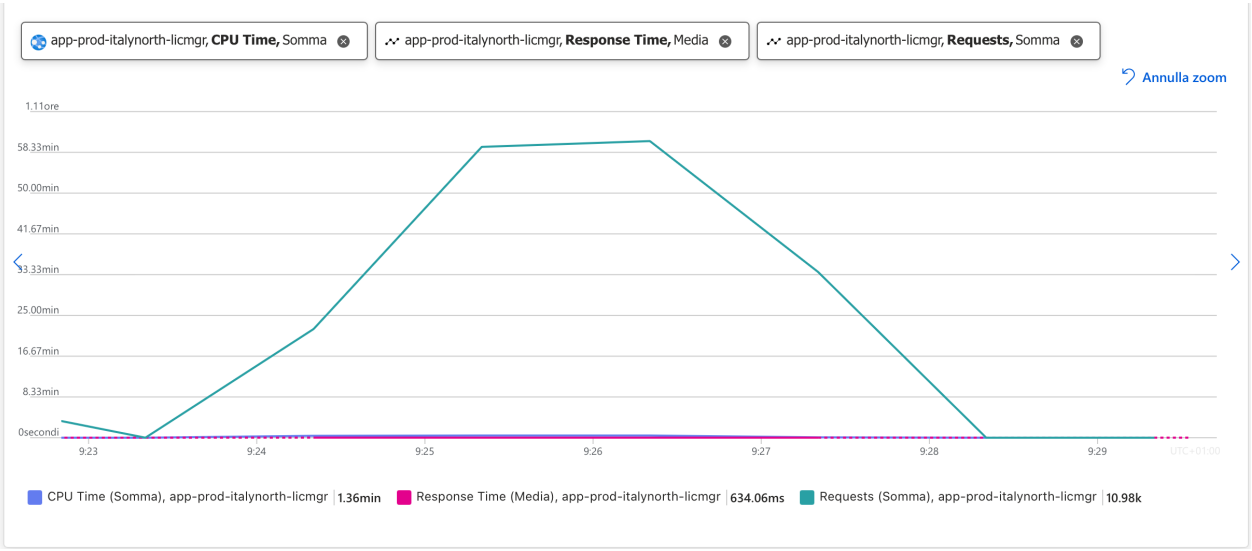

Figura 3.6: Grafico tempi di risposta e tempo CPU App Services

4. Analisi installazioni e dei cambiamenti aziendali

4.1 Prime installazioni effettuate

Si analizzeranno ora due delle prime installazioni effettuate tra Dicembre 2025 e Febbraio 2026. Verranno prese in considerazione una installazione a rete chiusa ed una a rete aperta. Le versioni utilizzate di GPOne differiscono in quanto una è stata fatta nei giorni immediatamente successivi al rilascio della versione con la nuova gestione licenze, mentre l'altra è stata effettuata circa due mesi dopo. In entrambe le installazioni sono stati coinvolti i reparti di consulenza, assistenza e ricerca e sviluppo. Il consulente ha effettuato l'aggiornamento in loco del software, l'assistenza in un caso e la ricerca e sviluppo in un altro sono intervenuti per modifiche ai moduli o al numero di abilitazioni della licenza.

4.1.1 Installazione con rete aperta

La prima installazione su rete aperta è stata fatta al cliente Overmach. La stessa Overmach che possiede la maggioranza delle quote di OSL è di fatto stata una dei primi clienti di GPOne. Il vantaggio di avere in casa un cliente in questo caso è stato il poterli monitorare quasi in tempo reale nei flussi di lavoro; questo ha permesso di ricevere feedbacks molto velocemente e di correggere eventuali bug nel giro di poche ore. Essendo di fatto una sorta di beta testers, si è deciso di provare sul loro server l'installazione del microservizio di controllo. Essendo già a conoscenza del portale online, il reparto IT di Overmach ha fornito in anticipo una regola di firewall per permettere la comunicazione con Azure. Il consulente, che in questo caso è anche analista, ha provveduto a installare la versione 1.2.0 di GPOne.

I passi che ha compiuto il consulente dal cliente sono stati i seguenti:

- effettuare l'accesso al portale per ottenere il la chiave di attivazione della licenza 1.2.0 per Overmach;
- utilizzare la chiave nel programma d'installazione di GPOne;

- impostare l'orario della chiamata di controllo (orario notturno).

In questo caso particolare in cui la figura del consulente e dell'analista coincidono, anche il censimento e l'attivazione della licenza sono stati fatti dallo stesso consulente. Tutto il processo dal censimento all'installazione non è durato oltre l'ora.

Il caso di Overmach ha poi portato a provare anche l'attivazione da remoto di un modulo. Overmach non possiede molti moduli attivi; ciò è dovuto al fatto che lavorano sulla personalizzazione di macchine utensili, ma a sistema le macchine non sono risorse, ma commesse; le uniche risorse sono gli operai. Non avendo risorse macchina, molti moduli sarebbero inutili. Gli unici moduli con un numero di abilitazioni che hanno sono quello Base per gli utenti e quello Fdc per i tablet degli operatori. Nel corso però dell'anno 2026 è previsto che si inizi a impostare una gestione magazzino più organizzata; questo dovrà passare anche per il modulo di gestione magazzino presente su GPOne. Per effettuare una prima formazione, soprattutto per raccogliere feedback, si è deciso di attivare a distanza il modulo Warehouse-Management. In questo caso l'analista ha semplicemente dovuto attivare il modulo sulla licenza di Overmach, impostandone la data di scadenza a fine gennaio. La licenza è stata aggiornata nella notte e nella giornata successiva si è potuta inizare la formazione.

4.1.2 Installazione con rete chiusa

L'installazione su rete chiusa è stata effettuata presso un cliente nella provincia di Torino. Il cliente in questione ha una gestione un po' più evoluta rispetto ad Overmach, in quanto presenta non solo più moduli per GPOne ma sfrutta anche altri prodotti aziendali (GPOne Green). L'installazione della versione precedente aveva avuto una serie di richieste un po' più particolari del solito, in quanto i vari servizi singoli, RabbitMQ, SQL Server e l'istanza di GPOne, dovevano essere su dischi diversi della macchina virtuale. Per il momento non ci hanno concesso di comunicare con un server esterno alla rete aziendale dunque l'installazione è stata fatta offline. L'aggiornamento è stato effettuato dal consulente nell'azienda, ma è stato supervisionato dalla ricerca e sviluppo in quanto alcuni sviluppi che avevano richiesto necessitavano di un programmatore in fase di installazione. Rispetto al caso Overmach, il consulente si è dovuto fisicamente scaricare il file di licenza del cliente dal portale, e non ha dovuto impostare nulla per il servizio di controllo. La licenza, che sul server è stata ricodificata usando l'indirizzo MAC della prima scheda di rete della macchina virtuale, aveva

ancora il vecchio numero di utenti abilitati, a causa di una dimenticanza in fase di creazione della nuova versione della licenza. Questo, come si era già visto nei bug trovati, ha creato problemi nell'accesso inizialmente. Essendo una rete chiusa, si è semplicemente fatto girare nuovamente il programma d'installazione con il file di licenza aggiornato. Seppure molto meno efficiente, i clienti che presentano questa situazione di solito sono anche quelli la cui offerta commerciale è più alta.

4.2 Risultati rispetto ai requisiti e feedback dei reparti

La prima iterazione dello sviluppo dell'applicazione delle licenze si è conclusa con le prime installazioni presso cliente. Rispetto ai requisiti di partenza, si può dire che essi siano stati rispettati tutti.

I requisiti funzionali e non erano principalmente legati allo sviluppo dell'applicazione in cloud (tabella 1.1 e 1.2). Sfruttando i servizi di Azure si è riusciti con uno sforzo non eccessivo a rispettare i requisiti RF02, RF03, RF07. Il requisito RF09 è stato rispettato per le installazioni con chiave di attivazione, mentre per il caso di rete chiusa si è comunque riusciti a mitigare il problema delle macchine virtuali utilizzando l'indirizzo MAC della scheda di rete. Il requisito RF08 è stato sostanzialmente aggirato fornendo agli analisti l'accesso al server SQL dell'applicazione licenze, senza dover predisporre un sistema di reportistica usando librerie terze; nel caso fosse necessario si potrà sempre aggiungere in un'iterazione futura.

Per i requisiti non funzionali RNF04 e RNF05, la scalabilità, le prestazioni e la disponibilità sono garantiti dall'utilizzo delle strutture di Azure e dalla struttura dell'applicazione. Nulla vieta di estendere l'applicazione licenze in un futuro a una gestione interna più grande, magari integrando anche una parte di gestione dipendenti, di gestione presenze e ferie, di gestione ordini e magazzino. In quell'ottica, l'applicazione licenze si può trasformare in un microservizio a se, magari containerizzandola, che si integrerà in un gestionale più elaborato.

I reparti in generale sono stati tutti abbastanza soddisfatti di questo primo sviluppo. I reparti predisposti alla creazione e approvazione licenza hanno trovato molto più veloce e intuitivo l'utilizzo del portale al posto del vecchio software, in quanto sviluppata per l'esigenza dell'utilizzatore. I reparti coinvolti nell'installazione da cliente hanno espresso tutti un'efficienza migliorata dal non dover passare tramite la vpn aziendale sul vecchio gestionale o tramite una chiamata all'assistenza

per ottenere il file di licenza. Il reparto commerciale ha sicuramente trovato una struttura di moduli e licenze molto semplificata rispetto a prima. Infine, il reparto assistenza e quello ricerca e sviluppo hanno una richiesta inferiore di problemi e richieste relative al passaggio di file di licenza o di aggiornamento dei moduli.

Conclusioni

La progettazione e l'implementazione dell'applicazione per la nuova gestione delle licenze ha soddisfatto le aspettative e le richieste di tutti gli stakeholders. L'applicazione è stata resa operativa da Novembre 2025 e integrata ufficialmente nei flussi aziendali dalla versione 1.2.0 di GPOne.

Benefici portati all'azienda

Il risultato più importante è stato lo snellimento e l'efficientamento dei flussi di rilascio licenza aziendali. Questo ha portato a un abbassamento dei tempi per eseguire gli aggiornamenti licenza, riducendo l'overhead dovuto alle comunicazioni tra reparti. A livello di sviluppo ha aperto per l'azienda la possibilità di utilizzare lo sviluppo in cloud per alcuni prodotti che verranno rilasciati in futuro. L'aver acquisito a livello aziendale e di reparto competenze in ambito cloud amplia anche la ricerca di personale a figure più eterogenee rispetto al classico programmatore fullstack.

Miglioramenti futuri sull'applicazione licenze e GPOne

Si è già accennato nelle sezioni precedenti all'evoluzione che si potrà in futuro ottenere dall'applicazione licenze. Non necessariamente l'applicazione si limiterà a gestire le licenze per GPOne, ma potrà essere estesa alla gestione delle licenze per tutti i prodotti aziendali e a gestire altri aspetti dell'azienda, andando a sostituire a lungo andare GP9Over per la gestione interna. Per GPOne difficilmente si potrà passare a una gestione completamente in cloud, data la natura del software; tuttavia è comunque possibile pensare a una futura installazione ibrida, in cui le anagrafiche vengono gestite in cloud e i collegamenti macchina e i pannelli vengono gestiti come dispositivi edge. Per gli ambienti di staging e test si può però pensare di spostare il tutto su una gestione su Azure, con ogni microservizio e client gestito con un App Services.

Ringraziamenti

Ringrazio innanzitutto l'azienda OSL S.r.l. che mi ha dato l'opportunità di portare il progetto che ho esposto ma che soprattutto da ormai 4 anni mi da l'opportunità di crescere e migliorare costantemente sia in ambito lavorativo che come persona. Ringrazio tutti i miei colleghi, della sede di Spilamberto e di Parma, in particolare il team di GPOne e tutti quelli facenti parte della R&D, che hanno vissuto in prima persona tutto il mio viaggio per prendere questa laurea, i momenti difficili in pausa pranzo a studiare e i momenti di euforia a seguito degli esami passati, ci andremo a mangiare del sushi dopo la laurea ve lo prometto.

Ringrazio il professore Riccardo Lancellotti che si è preso in carico di seguirmi abbastanza all'improvviso nella realizzazione di questa tesi e che nonostante il poco tempo che ha avuto a disposizione in questi mesi non mi ha fatto mai mancare feedback e consigli.

Ringrazio tutti i miei amici che mi continuano a sostenere da anni, i ragazzi del Vexismo anche se non ci si vede più così spesso, chi mi è vicino dai tempi delle elementari che ormai più che amici sembrano quasi dei fratelli.

Ringrazio i miei parenti sparsi ormai per il mondo (letteralmente), che mi sostengono da quando sono piccolo anche se ormai ci si vede soprattutto a natale. Ringrazio soprattutto chi ha fatto uno sforzo importante per poterci essere nel momento della cerimonia e questo ringraziamento va alla mia nonna Rosetta e a mia cugina Alessia che si sono messe nel treno da Torre Annunziata fino a qui.

Dedicherò un po' più parole a questi ultimi due ringraziamenti.

Rebby, ormai sono oltre 7 anni che condividiamo questo viaggio. I nostri percorsi specialmente negli ultimi 4 anni sono stati molto simili: abbiamo iniziato a lavorare più o meno nello stesso momento, abbiamo deciso di continuare gli studi e di continuare a lavorare, nonostante ci venisse detto che non era necessario stressarci così tanto pur di fare entrambe le cose. Ci siamo stressati, e pure tanto, pur di arrivare a questo obiettivo. Anche nei miei momenti più complicati, ho sempre pensato che se ce la stavi facendo tu, che secondo me hai affrontato un percorso più pesante, io non potevo essere da meno. Io ti ringrazio, sia per il supporto che mi hai dato direttamente che per la motivazione che mi hai dato indirettamente; e questo non solo nei miei momenti di down nello

studio e nel lavoro ma anche in qualsiasi cosa facessi, dagli obiettivi improbabili di correre mezze maratone a quelli di comprare chitarre un tantino costose (non dico quanto ho speso che poi quelli dei ringraziamenti successivi si arrabbiano, ma sai bene che parlo di quella di JP). Non negherò che mi sarebbe piaciuto batterti e laurearmi prima di te anche di un solo giorno pur di prenderti un po' in giro, ma mi accontento di un dignitoso secondo posto. Voglio continuare a viaggiare insieme a te anche nei prossimi passi delle nostre vite, a suon di chitarre, pietrine, libri e funko pops.

Mamma e papà, ve lo dirò con il cuore in mano, siete stati leggermente dei rompiscatole in questi 4 anni, anche se in realtà potrei estenderlo a tutti i miei 27 anni di vita. Vi vorrei far notare che, se uno non studia alle 7 di sera post lavoro per gli esami o non passa tutto il tempo a scrivere la tesi non è perchè non studia e non ha voglia di fare nulla ma forse, e dico forse, è perchè è un po' stanco dalla giornata lavorativa. Tolto questo sassolino passiamo alle cose serie. Tutto il percorso universitario, e non solo, se l'ho potuto fare e completare lo devo soprattutto grazie a voi due, voi che per mille motivi avete affrontato una vita molto più complicata della mia, che avreste voluto fare altro, che avreste voluto anche voi aver l'occasione di andare all'università, avere altre opportunità e ve lo sareste pure meritato per tutti i sacrifici che avete fatto e che continuate a fare. Questo successo è vostro tanto quanto è mio. Sapete già, perchè già la state affrontando, che la parte successiva delle nostre vite sarà complicata, non potrete più essere la base solida che mi ha aiutato ad attraversare questi 27 anni. Toccherà a me fare il possibile per essere la vostra base solida, per restituire in qualche modo tutto quello che mi avete dato in questi anni, idealmente anche di più. Adesso possiamo festeggiare coi coriandoli che non abbiamo lanciato alla triennale.

Bibliografia

[1] ACN. *NIS - Network Information Security*. Portale dell'ACN per la NIS. 2025. URL: https://www.acn.gov.it/portale/nis.

[2] Medium - Sehban Alam. *What is Zone.js in Angular*. Spiegazione di Zone.js in Angular. 2026. URL: https://medium.com/@sehban.alam/what-is-zone-js-in-angular-e0029c21c32f.

[3] Angular. *Signals Overview Angular*. Documentazione dei signals di Angular. 2026. URL: https://angular.dev/guide/signals.

[4] Augeos. *Normativa NIS2: come capire se azienda coinvolta e azioni da fare*. Spiegazione della NIS2 per le aziende. 2024. URL: https://blog.augeos.it/normativa-nis2-come-capire-se-azienda-coinvolta-e-azioni-da-fare.

[5] Nicola Bicocchi. *Software Architectures*. Spiegazione delle varie architetture software. 2025. URL: https://github.com/nbicocchi/learn-microservices/blob/main/modules/introduction/slides/3%20-%20Software%20architectures.md.

[6] GDPRScuola. *I nuovi diritti degli utenti con il GDPR*. I nuovi diritti degli utenti con il GDPR. 2025. URL: https://magazine.gdprscuola.it/articoli/i-nuovi-diritti-degli-utenti-con-il-gdpr/.

[7] Guardey. *Qual'è la differenza tra NIS1 e NIS2?* Spiegazione differenze tra le due normative. 2024. URL: https://www.guardey.com/it/what-is-the-difference-between-nis1-and-nis2/.

[8] IBM. *Autenticazione JWT*. Documentazione dell'autenticazione JWT. 2025. URL: https://www.ibm.com/docs/it/order-management?topic=users-jwt-authentication.

[9] IBM. *Cos'è GraphQL?* Spiegazione della tecnologia GraphQL. 2025. URL: https://www.ibm.com/it-it/think/topics/graphql.

[10] Inc. Massient. *MassTransit*. Documentazione MassTransit. 2025. URL: https://masstransit.io.

[11] MetaCompliance. *Guida ai 7 principi chiave del GDPR*. Guida ai 7 principi chiave del GDPR. 2025. URL: https://www.metacompliance.com/it/blog/governance-rischio-conformita-grc/gdpr-guida-ai-7-principi-chiave.

[12] Microsoft. *Documentazione di Azure*. Documentazione di Azure. 2025. URL: https://learn.microsoft.com/it-it/azure/.

[13] Microsoft. *Event-Driven Architecture Style*. Spiegazione architettura event-driven. 2025. URL: https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/event-driven.

[14] Microsoft. *Panoramica di ASP.NET Core SignalR*. Documentazione di Microsoft SignalR. 2025. URL: https://learn.microsoft.com/it-it/aspnet/core/signalr/introduction?view=aspnetcore-10.0.

[15] Microsoft. *Why Azure vs. AWS*. Confronto tra Azure e AWS. 2025. URL: https://azure.microsoft.com/en-us/pricing/azure-vs-aws.

[16] Normattiva. *DECRETO LEGISLATIVO 4 settembre 2024, n.138*. Decreto legge NIS2. 2024. URL: https://www.normattiva.it/uri-res/N2Ls?urn:nir:stato:decreto.legislativo:2024-09-04;138!vig=.

[17] PrimeNG. *PrimeNG - Angular UI Component Library*. Documentazione di PrimeNG/Sakai. 2025. URL: https://primeng.org.

[18] ZZZ Projects. *What is Entity Framework?* Documentazione Entity Framework Core. 2025. URL: https://www.entityframeworktutorial.net/entityframework6/what-is-entityframework.aspx.

[19] RabbitMQ. *RabbitMQ: One broker to queue them all*. Documentazione RabbitMQ. 2025. URL: https://www.rabbitmq.com.

[20] GitHub - RicoSuter. *NSwag: the Swagger/OpenAPI toolchain for .NET, ASP.NET Core and TypeScript.* Documentazione NSwag. 2026. URL: https://github.com/RicoSuter/NSwag.

[21] Chariot Solutions. *Http and Observable in Angular2 . how to work asyncronously*. Spiegazione degli observables in Angular2. 2025. URL: https://chariotsolutions.com/blog/post/angular2-observables-http-separating-services-components/.

[22] O.S.L. Srl. *Chi siamo*. Informazioni O.S.L. Srl. 2025. URL: https://www.osl.it/chi-siamo/.

[23] Wikipedia. *Regolamento generale sulla protezione dei dati*. Regolamento generale sulla protezione dei dati. 2025. URL: https://it.wikipedia.org/wiki/Regolamento_generale_sulla_protezione_dei_dati.